ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA DEL SOFTWARE

SUMMON STORM

PROGRAMACIÓN Y DISEÑO DEL VIDEOJUEGO E INTELIGENCIA ARTIFICIAL

SUMMON STORM

VIDEO GAME DESIGN, PROGRAMMING AND ARTIFICIAL INTELLIGENCE

Realizado por
Carlos de Miguel Platero
Tutorizado por
Enrique Domínguez Merino
Coordinado por
Enrique Domínguez Merino
Departamento
Lenguajes y ciencias de la computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Junio de 2017

Fecha defensa:
El Secretario del Tribunal

Resumen:
El trabajo consiste en el diseño e implementación de un videojuego de estrategia por turnos, el resultado se tratará, por tanto, de un producto de entretenimiento, por lo que se le ha dado gran importancia a la experiencia de usuario, priorizando la usabilidad del sistema. Por ello es clave la interfaz y el entorno 3D en el que se desarrolla el juego.

El juego permitirá a dos usuarios enfrentarse en línea o en un mismo ordenador, y también a un sólo usuario enfrentarse a un agente inteligente. Esto supone la implementación de una Inteligencia Artificial que suponga un cierto reto, sin llegar a ser invencible, estudiando diferentes estrategias con el fin de conseguir una óptima relación entre el tiempo empleado en un turno y la calidad de la acción elegida para ejecutar en dicho turno.

La implementación del sistema se hará sobre el motor de videojuegos Unity y el marco de trabajo que proporciona, en el lenguaje de programación C#.
Para la programación se han usado la arquitectura y los patrones de diseño de software necesarios para asegurar la correcta implementación del sistema, así como su posterior funcionamiento y flexibilidad.

Palabras clave: videojuego, Unity, C#, inteligencia artificial, estado de juego, búsqueda de estados, juego en línea, juego.

Abstract:
The goal of the project is the design and the implementation of a strategy turn-based video game. The purpose of our work is to elaborate an entertainment product, that is the reason why the user experience has been treated as the main aspect of the application, establishing the usability of the system as the highest priority. Because of this, the interface and the 3D graphic environment in which the game is played are the key points of the system.

The game will allow two users to play online or on the same computer, allowing also a single user to play against the computer. This means the implementation of an Artificial Intelligence that involves a certain challenge, without being invincible, studying different strategies in order to achieve an optimal ratio between the time spent and the quality of the action.

The implementation of the system will be done based on the video game engine Unity and the framework it provides, in the programming language C#.
In coding terms, we have used the system architecture and the software design patterns that are needed to ensure the proper implementation and the correct operation and flexibility of the system.

Keywords: video game, Unity, C#, Artificial Intelligence, game state, search algorithm, online game, game.

Índice general

Índice de figuras

1. Introducción

1.1. Motivación

Nos encontramos en un momento en el que la industria del entretenimiento que más está creciendo es la de los videojuegos, lo que hace que también genere mucho empleo. El desarrollo de videojuegos es altamente multidisciplinar, abarcando desde las matemáticas y ramas de la física como la mecánica, a disciplinas artísticas como el dibujo o la música. Una de estas disciplinas, y siendo además de las más importantes, puesto que el videojuego es un programa informático , es el desarrollo de software.[1]
También es interesante indicar que se está fomentando el uso de los conocimientos del ámbito de los videojuegos en otras aplicaciones del software, para ofrecer mejores experiencias al usuario, difuminando la línea entre las aplicaciones de ocio y de trabajo/aprendizaje. Y este concepto, llamado gamificación, está dando grandes resultados, sobretodo en el entorno educativo, aumentando el rendimiento de los alumnos.

1.2. Objetivos

Hemos decidido crear un videojuego, abarcando a la vez gran parte de las asignaturas del grado y gran parte de las aptitudes necesarias para programar un juego completo. De forma que el trabajo será una aplicación de los conocimientos provenientes del grado a la implementación de un videojuego tanto para jugar en línea, para lo que se usarán bases de datos, conexiones cliente/servidor, etc., como para jugar contra el ordenador, aplicando conocimientos de Inteligencia Artificial, y aplicando conceptos como la programación estructurada, el análisis de algoritmos o los patrones de diseño, a la programación del videojuego. El objetivo final es la programación del videojuego, con todo el aprendizaje sobre programación que ello conlleva, pero también existen otros subobjetivos en este Trabajo de Fin de Grado tales como el análisis de requisitos del producto a desarrollar, el uso de metodologías ágiles en grupo, o la evaluación de los resultados obtenidos, que han sido muy enriquecedores para nuestra formación

Debido a la complejidad del objetivo considerado, se decidió realizar este Trabajo Fin de Grado en la modalidad de grupo. Ese objetivo principal lo hemos dividido en tres partes: la común, la desarrollada en este proyecto y la desarrollada por mi compañero Manuel Álvarez, que se ha encargado del juego en línea. De forma breve, la parte común a consistido en el

análisis del producto a desarrollar, y mi parte ha consistido en la programación del juego sobre el motor gráfico **Unity** y la posterior implementación de la Inteligencia Artificial.

1.3. Entorno tecnológico

La herramienta principal de trabajo ha sido el motor de videojuegos **Unity5**[2] y el entorno de programación **Visual Studio**. Para la elaboración de la memoria se ha usado el lenguaje de maquetación LaTeX [3], en concreto usando la herramienta **TexStudio**.

1.3.1. Unity5

Un motor de videojuegos que ofrece un marco de trabajo potente para simular físicas, detectar colisiones, manejar modelos 3D y 2D, y ejecutar animaciones y efectos gráficos y sonoros. Sin embargo, nuestro juego, por sus particularidades, no se ha visto demasiado apoyado en las funciones nativas de **Unity5**, sino que se ha tenido que programar tanto la lógica interna del juego, como las mecánicas de juego base, de forma casi completa, y, una vez implementadas, apoyarse en dichas funciones de nativas de *Unity5* para mostrar por pantalla un entorno agradable en 3D que simule el estado de juego, y capte las interacciones con este. Por tanto, no fue necesario estudiar en profundidad la programación sobre *Unity5* hasta el capítulo **Entorno gráfico e interfaz**, por ello, será al inicio de la sección cuando se explicará lo necesario para poder continuar con la implementación en ese sentido.

1.3.2. Desarrollo e implementación

EL total del código ha sido escrito en lenguaje **C#**[4] en el entorno de desarrollo integrado **Visual Studio**

1.3.3. Herramientas de modelado

Para los modelos previos a la implementación, se ha usado la herramienta **Magic Draw**, para los diagramas finales de implementación, se han usado los diagramas de clases que **Visual Studio** infiere del código, proporcionando así diagramas totalmente consistentes con lo programado. Gracias a esto, a lo largo de esta memoria, se podrá observar cómo difieren los diagramas iniciales y finales, así como, a pesar de que los diagramas finales, tras la implementación real son mucho más complejos, la estructura sigue siendo la misma que la estudiada en la fase de modelado.

1.4. Metodología de trabajo

Al ser un trabajo en grupo, ha sido necesaria la manipulación simultánea del proyecto, tanto para poder aportar bilateralmente nuevas funcionalidades al proyecto, como para que lo que cada alumno vaya haciendo esté disponible para el otro alumno de la mejor manera posible y

más rápida. **Unity5** proporciona una herramienta, **Unity Collaborate**, que permite a grupos, de forma gratuita, compartir y sincronizar proyectos, haciendo las veces de repositorio y de sistema de control de versiones. [5]

La parte de gestión tecnológica queda cubierta, por tanto, con esta herramienta.
Para la parte de gestión del proyecto en cuestiones de planificación, división de tareas y retrospectiva, hemos usado una metodología ágil, *Scrum*, que hemos adaptado a nuestras necesidades, basándonos, eso sí, en los principios de la filosofía de trabajo ágil. Hemos realizado reuniones semanales en las que se descomponían los requisitios más prioritarios en tareas, y de las cuales decidíamos cual íbamos a desarrollar hasta la siguiente reunión, sin embargo no siempre han podido ser estrictamente semanales, debido a nuestras necesidades de dedicar más o menos tiempo a otras cuestiones universitarias y de empleo. Pero no ha sido un problema, precisamente por la naturaleza flexible de la metodología *Scrum*.

1.5. Estructura de la memoria

La memoria se ha estructurado de forma análoga al desarrollo, los capítulos muestran los principales objetivos del trabajo, y se han desarrollado en el mismo orden en el que se narran.

- Primero se efectuó un proceso previo a la implementación, siguiendo las técnicas propias de la Ingeniería del Software, cuyos resultados se recogen en el capítulo **Análisis del juego** y en los anexos I y II. Este capítulo representa la parte común a ambos alumnos, anteriormente mencionada.
- Una vez hecho esto, nos encontramos con la **Diseño del juego** que establece la estructura de datos y unas primeras funciones, en las que se basa todo el sistema.
- Posteriormente, el capítulo **Entorno gráfico e interfaz** narra cómo se ha construido un entorno 3D sobre esa lógica interna.
- En el siguiente capítulo, **Interacción con el usuario**, se explica cómo se ha integrado al usuario en un sistema tan flexible en términos de interacción y modos de uso como este.
- En el capítulo **Inteligencia Artificial**, que muestra el proceso de desarrollar un agente inteligente que permita ofrecer al usuario un enemigo virtual contra el que enfrentarse.
- Terminada la implementación, en el último capítulo, **Conclusiones** se han comentado de forma breve los aspectos más interesantes del trabajo realizado.

2. Análisis del juego

2.1. Definición de las reglas de juego y producto final

El primer paso de la creación del juego se trata de definir de forma clara cuáles son las reglas de este, cuántos jugadores participan, cómo lo hacen, y quién gana. Se ha redactado un documento que define el juego a obtener como producto final, sin embargo, este traspasa las limitaciones de tiempo y esfuerzo de un trabajo de estas características y, debido a ello, se ha establecido que el trabajo debería tener como resultado un producto mínimo viable. El proceso de captación y análisis de requisitos, así como la estructuración del código y el modelado, se han hecho para el producto total, del que luego se han implementado las partes más prioritarias que funcionen como producto mínimo viable, siendo este el producto real del trabajo.

La redacción producto de este proceso, se encuentra en el **Anexo I: Reglas del juego**. No obstante en esta sección se pretende hacer un resumen que permita el entendimiento de las reglas del juego.

El juego simulará una batalla entre dos jugadores sobre un tablero de casillas hexagonales. En dicho tablero, los jugadores estarán representados con unas unidades principales llamadas Héroes, las cuales definen el objetivo de la partida: eliminar al Héroe rival para poder ganar la partida.
Mediante los héroes, ambos jugadores podrán invocar nuevas unidades sobre el tablero para que le ayuden a derrotar al rival. No obstante existen ciertas limitaciones para estas unidades invocadas: Puede haber un máximo de 6 unidades, además del héroe, a la vez en el tablero por cada jugador, así como también cada héroe tendrá un número limitado de unidades totales que podrá invocar (este límite puede variar dependiendo del héroe).
Estas unidades cuentan con las siguientes características:

-*Vida*: Al llegar a 0 la unidad muere.

-*Ataque*: Puntos de vida que restará a una unidad enemiga al atacar.

-*Defensa*: Daño que evita la unidad al recibir un ataque

-*Rango de Ataque*: Distancia a la cual una unidad puede realizar un ataque.

-*Rango de Movimiento*: Distancia por la que se podrá mover como máximo durante un turno.

Estas unidades, así como el héroe, pueden realizar una serie de acciones que vendrán en cierta medida delimitadas por algunas de las características mencionadas. Estas acciones serían:

- Movimiento: Mueve una unidad deseada desde la casilla en la que se encuentra, a otra casilla dentro de su rango de movimiento.

- Ataque: Inflige daño a un rival dentro de su rango de ataque, para reducir su vida una cantidad igual a sus puntos de ataque. Si la unidad rival posee puntos de defensa, serán estos los que desciendan una cantidad igual al daño infligido. Si la unidad posee puntos de defensa, pero recibe un daño superior a dichos puntos, la diferencia de puntos será obviada y dejará a la unidad sin puntos de defensa, pero sin reducir sus puntos de vida.

Por otro lado, estas unidades pueden hacerse más fuertes mediante evoluciones, al derrotar unidades enemigas. Es decir, cuando cualquier unidad propia derrota a una unidad rival, dicha unidad obtiene una cantidad de puntos de evolución establecidos por la propia unidad derrotada. Estos puntos de evolución permiten evolucionar cualquier unidad propia sobre el tablero, mientras se tengan los puntos necesarios.
Es necesario indicar también, que cada jugador podrá realizar una única acción por turno, es decir, en cada turno un jugador únicamente podrá atacar con una unidad, mover una unidad, invocar una unidad o evolucionar una unidad, así como también podrá decidir no realizar ninguna acción y ceder el turno al rival. Ceder el turno al rival no tiene ningún beneficio para el jugador que lo realiza.

Con estas reglas ya se puede comprender el funcionamiento del juego. Por otro lado, como se ha mencionado al principio de esta sección, en el **Anexo I: Reglas del juego**, se encuentran las reglas del juego completamente detalladas.

2.2. Análisis de requisitos

A la hora de analizar los requisitos, se utilizaron tanto las reglas de juego, como las historias de usuario.

Historias de usuario:

Las historias de usuario a alto nivel consisten en los posibles modos de juego para el usuario, que son:

- Jugar contra otro jugador en línea.
- Jugar contra otro jugador de manera local.
- Jugar contra una Inteligencia Artificial.

La redacción de los requisitos en profundidad se puede encontrar en el **Anexo II: Análisis de requisitos**.

3. Diseño del juego

3.1. Elementos de juego y estructura de datos

Este es el diagrama clases establecido en la fase de modelado para almacenar los datos de la partida y especificar las funciones de la lógica interna de la aplicación:

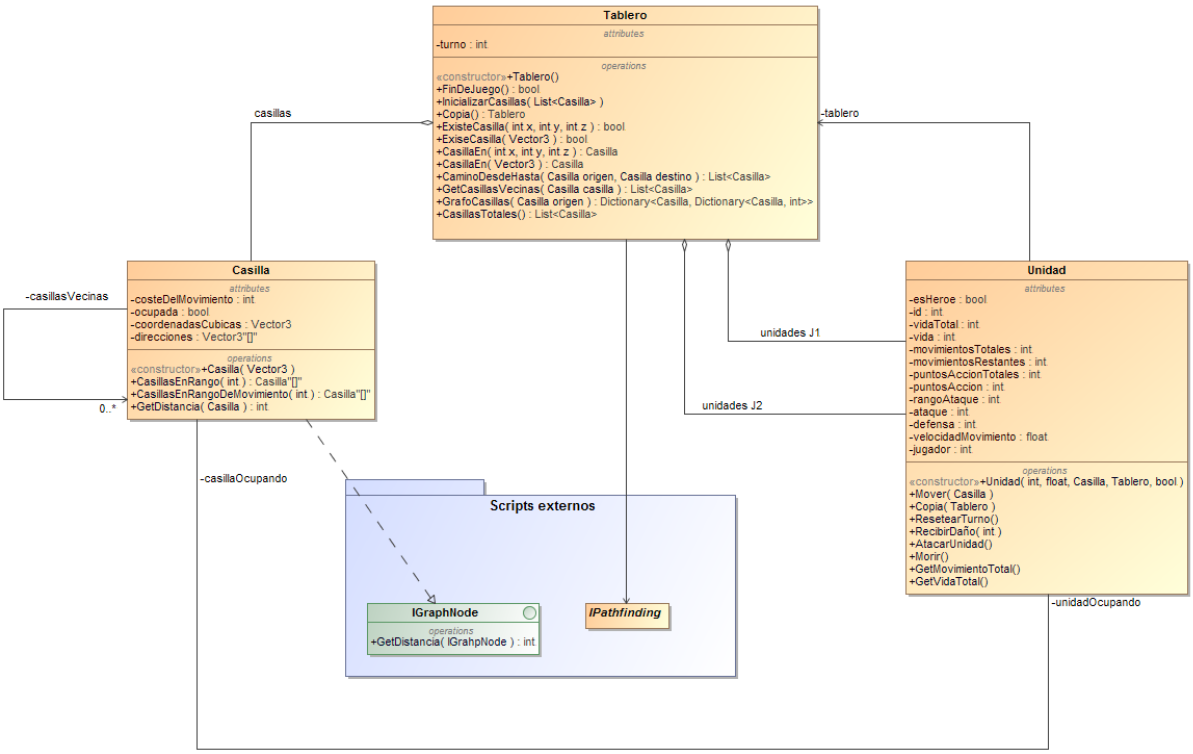
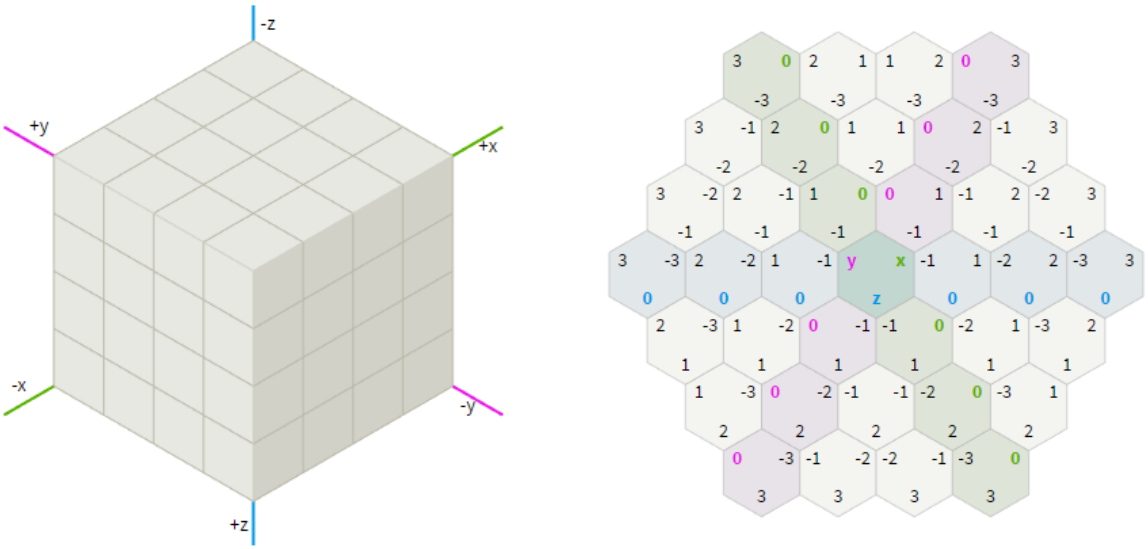


Figura 3.1: Diagrama pre-implementación de la lógica interna

A continuación se enumeran cada una de estas clases y sus puntos más interesantes:

- Casilla:
  Son las casillas que componen el tablero, nos sirven para posicionar y mover por ellas las unidades Al tratarse de casillas hexagonales, para su tratamiento manejamos 3 coordenadas, teniendo en cuenta X,Y,Z. Trasladando este tipo de coordenadas usadas para representar 3 dimensiones, al conjunto de formas hexagonales.



(a) Modelo en 3 dimensiones

(b) Transposición al plano[6]

Figura 3.2: Modelo de coordenadas para las casillas

Como se puede observar, para cada dirección se aumenta o disminuye la coordenada correspondiente, quedando a 0 siempre en la dirección transversal. Sin embargo, al usar 3 coordenadas en un entorno plano, sobran conjuntos (X,Y,Z), y para que no puedan darse dos conjuntos distintos que hagan referencia a la misma casilla, se establecerá que el sumatorio de las tres coordenadas será siempre igual a cero, como se puede ver 4.5a.

Agente externo: Pathfinder

También se puede ver en el diagrama de la figura 3.1, cómo la casilla implementa la interfaz **IGraphNode**, la cual especifica el método para obtener la distancia a otro IGraphNode, esto es necesario ya que se ha importado un agente externo de la comunidad oficial de *Unity5* , llamado *Pathfinder*[7], este agente usa un algoritmo A* para encontrar el camino entre dos elementos que implementen la interfaz **IGraphNode**, usando como heurística la distancia total entre dos nodos.
El resultado del pathfinder será el camino, salvando obstáculos, entre dos casillas, y se usará para señalar al usuario que camino puede seguir una unidad para moverse, así como para realizar las animaciones de movimiento de las unidades. Este método para obtener la mínima distancia entre dos casillas se ha implementado basado en la estructura de datos de coordenadas cúbicas, ya que gracias a esta se puede calcular de forma eficiente y sencilla la distancia, sin considerar obstáculos, entre dos casillas. De la siguiente forma:

```
public int GetDistancia(Casilla destino){
int distancia = (int)(Mathf.Abs(coordenadasCubicas.x -
destino.coordenadasCubicas.x) + Mathf.Abs(coordenadasCubicas.y -
destino.coordenadasCubicas.y) + Mathf.Abs(coordenadasCubicas.z -
destino.coordenadasCubicas.z)) / 2;
return distancia;
}
```

- Unidad:
  Las unidades son el activo principal de cada jugador. Las puede usar para derrotar al enemigo contrario ganando así el juego. Muchas de las acciones del jugador recaen en funcionalidades implementadas en la clase **Unidad**, como pueden ser **Mover**, **AtacarUnidad**, o **RecibirDaño**, que se ven en esta clase dentro del diagrama de la figura 3.1.

  Además, estas unidades tienen unos datos estáticos que especifican la naturaleza de la unidad, como puede ser cual es el jugador dueño de ella, el tipo de la unidad, la vida total, o el ataque de esta, y datos dinámicos, como la vida y defensa restantes o los movimientos que puede ejecutar en un determinado turno, incluyendo dentro de estos datos dinámicos, la casilla sobre la que está, como se ve en la relación **casillaOcupando**, siendo a su vez esta unidad la que se almacena en el campo **unidadOcupando** de la casilla.

  Existe un método, **ResetearTurno()**, que restablece los parámetros dinámicos pertinentes al iniciar el turno, como puede ser por ejemplo que sus movimientos restantes vuelvan a ser los movimientos totales en un turno de la unidad.

- Tablero:
  El tablero almacena el conjunto de casillas que lo componen, y las unidades que están en juego en ese momento, en distintas listas según su jugador dueño. El tablero almacena qué jugador tiene el turno en ese momento, y tiene un método **FinDeJuego()**, que devuelve verdadero en caso de que sea un estado de juego finalizado, y falso en otro caso.

  Un método interesante es **GrafoCasillas(Casilla origen)**, que devuelve el tablero en forma de grafo donde los nodos son casillas, siendo la casilla origen desde la que se llama al método el nodo raíz, y siendo los pesos de los arcos la distancia mínima entre ambos nodos **Casilla**. Este grafo es el usado para proporcionar al algoritmo A* encargado de establecer el camino entre casillas previamente mencionado, una estructura global sobre la que trabajar, disminuyendo la complejidad del cálculo mucho más que si simplemente se proporcionara una lista.

Después del proceso de implementación, se ha conseguido un código final bastante semejante al de la fase inicial de modelado, con añadidos como pueden ser funciones más específicas

que se han visto necesarias, o la reproducción de ciertos datos para mejorar la eficiencia. Además de ciertos campos y funciones para su integración con las representaciones físicas de las casillas y unidades en el entorno 3D, implementadas después de estudiar la arquitectura propia de *Unity5* para esto.

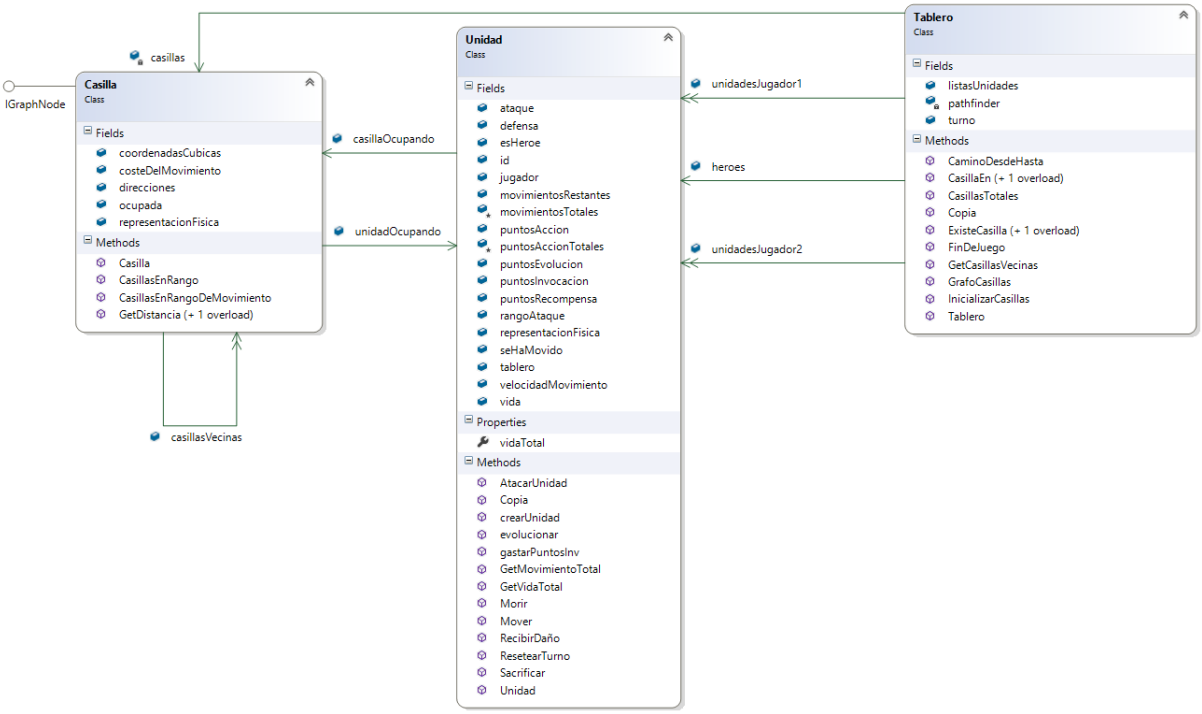


Figura 3.3: Diagrama final de la lógica interna

3.2. Estado de juego

Durante la implementación se manifestó el problema de que se concentraba una gran carga de funcionalidad en la clase tablero, ya que se había considerado al tablero en sí como el estado de juego, sin embargo eran necesarios más parámetros y funcionalidades, para esto se añadió al esquema de la lógica de juego la clase **EstadoJuego** y la clase **Acción**.

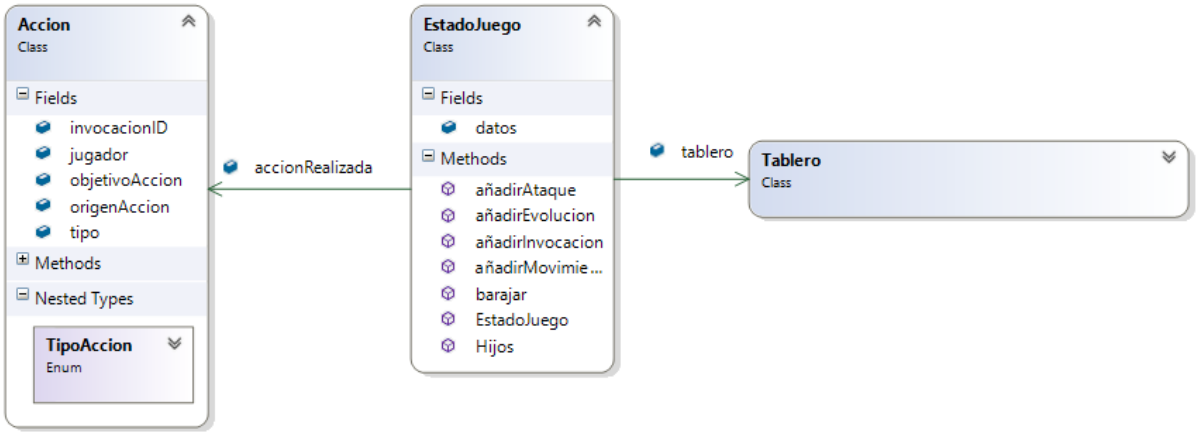


Figura 3.4: Diagrama final del estado de juego

De forma que la clase **Acción** nos dice cual ha sido la acción que nos ha llevado a ese estado, por ejemplo, si el tipo es movimiento, sabemos que en el último turno se realizó un movimiento de la unidad situada en la casilla **origenAccion**, hasta la casilla **objetivoAccion**, así como el jugador que lo realizó, almacenándose todos estos datos en propiedades de la clase **Acción**, como se ve en la figura 3.4.

Por otra parte, los métodos de **EstadoJuego**, nos permiten añadir una nueva acción al histórico de la partida, consiguiendo así un nuevo estado, donde la acción es la que se acaba de añadir, y el tablero es el resultado de aplicar al tablero anterior dicha acción. El método **Hijos** nos devuelve una serie de objetos de tipo **EstadoJuego**, siendo ellos todos los posibles estados a los que se puede llegar desde el estado en el que se ha ejecutado el método, con cada una de las posibles acciones que las reglas permitan y las restricciones del juego no prohíban.

Tanto a la hora de calcular los hijos de un estado de juego, como para que el servidor sepa detectar si un jugador ha enviado una acción que llevaría a un estado de juego ilícito, es necesario establecer las restricciones del estado de juego.

- Restricciones sobre el estado de forma estática

  La suma de las 3 coordenadas de cada casilla debe ser igual a 0

  $\forall c : Casilla | \{c.x + c.y + c.z = 0\}$

  Los atributos casillaOcupando y unidadOcupando son simétricos

$\forall(c : Casilla, u : Unidad) | \{c = u.casillaOcupando \Leftrightarrow c.unidadOcupando = u\}$

No puede haber una misma unidad en dos casillas

$\nexists (c1, c2 : Casilla) \mid \{c1.unidadOcupando = c2.unidadOcupando\}$

Todas las unidades deben tener vida extrictamente positiva y no superar su vida total

$\forall u : Unidad | \{u.vida > 0 \land u.vida \leq u.vidaTotal\}$

Cada jugador tiene como máximo 7 unidades en juego

$\forall t : Tablero \rightarrow (|j.listaJugador1| \leq 7 \land |j.listaJugador2| \leq 7)$

- Restricciones sobre las acciones

Estas restricciones se refieren a las reglas de juego, (de qué forma se pueden mover las unidades, qué pasa cuando se ejecuta un ataque, donde se puede invocar una unidad y qué es necesario para ello, etc.). Se podrían establecer formalmente a través de cada tipo de acción y sus precondiciones y postcondiciones, pero existen muchos casos especiales y además, las reglas descritas lo son para el producto mínimo viable y no para el producto final, por lo que para este apartado nos hemos basado en las reglas claramente definidas en el **Anexo I: Reglas del juego**.

4. Entorno gráfico e interfaz

En este capítulo se hablará de los elementos visuales del juego, los cuales se han ido creando bajo demanda, es decir, una vez implementado un subconjunto independiente de la lógica interna necesaria para que estos elementos se puedan representar correctamente, se les añade a esta lógica lo necesario para enlazarlos con sus representaciones físicas y poder acceder a todos los datos que sean necesario visualizar.

4.1. Conceptos previos

Para la representación gráfica del juego se han usado las funcionalidades y estructura que proporciona *Unity5* y los modelos se han creado con el software **Blender**

4.1.1. GameObject y MonoBehaviour

Unity5 proporciona un entorno gráfico muy potente, y para trabajar sobre él ofrece una arquitectura propia que dota de libertad total al programador.
Para ofrecer al usuario una visualización de objetos de juego en pantalla son necesarias dos cosas: El objeto gráfico y el comportamiento de éste, siendo el objeto gráfico un elemento que necesita un diseño tanto estético como funcional, y siendo el comportamiento las funciones que se implementan para este objeto.

Este motor ofrece también una serie de componentes que se pueden acoplar al objeto de juego, dotándolo de una funcionalidad preestablecida, pero para que podamos programar libremente su comportamiento, ofrece una clase propia, *MonoBehaviour*[8], que especifica un comportamiento común, de ahí el nombre, para los objetos que se representan gráficamente sobre la escena de juego. De forma que podemos programar una clase que hereda de *MonoBehaviour* y luego añadir un componente de esta clase al objeto de juego, y así este objeto se comportará según la implementación de dicha clase.
En la figura 4.1 se puede observar cómo los objetos de juego, *GameObject* , son una composición de componentes, y que se ofrece una clase, *MonoBehaviour*, que es a su vez un componente programable. Además se pueden ver otros componentes predefinidos que usamos en nuestro proyecto, como pueden ser *cámaras* que ofrecen funcionalidades para renderizar la escena, *gestores de colisiones*, que nos permiten manejar colisiones en el entorno 3D, o el componente *Transform*, que se usa para solicitar o manipular la posición de un objeto dentro

del entorno 3D.

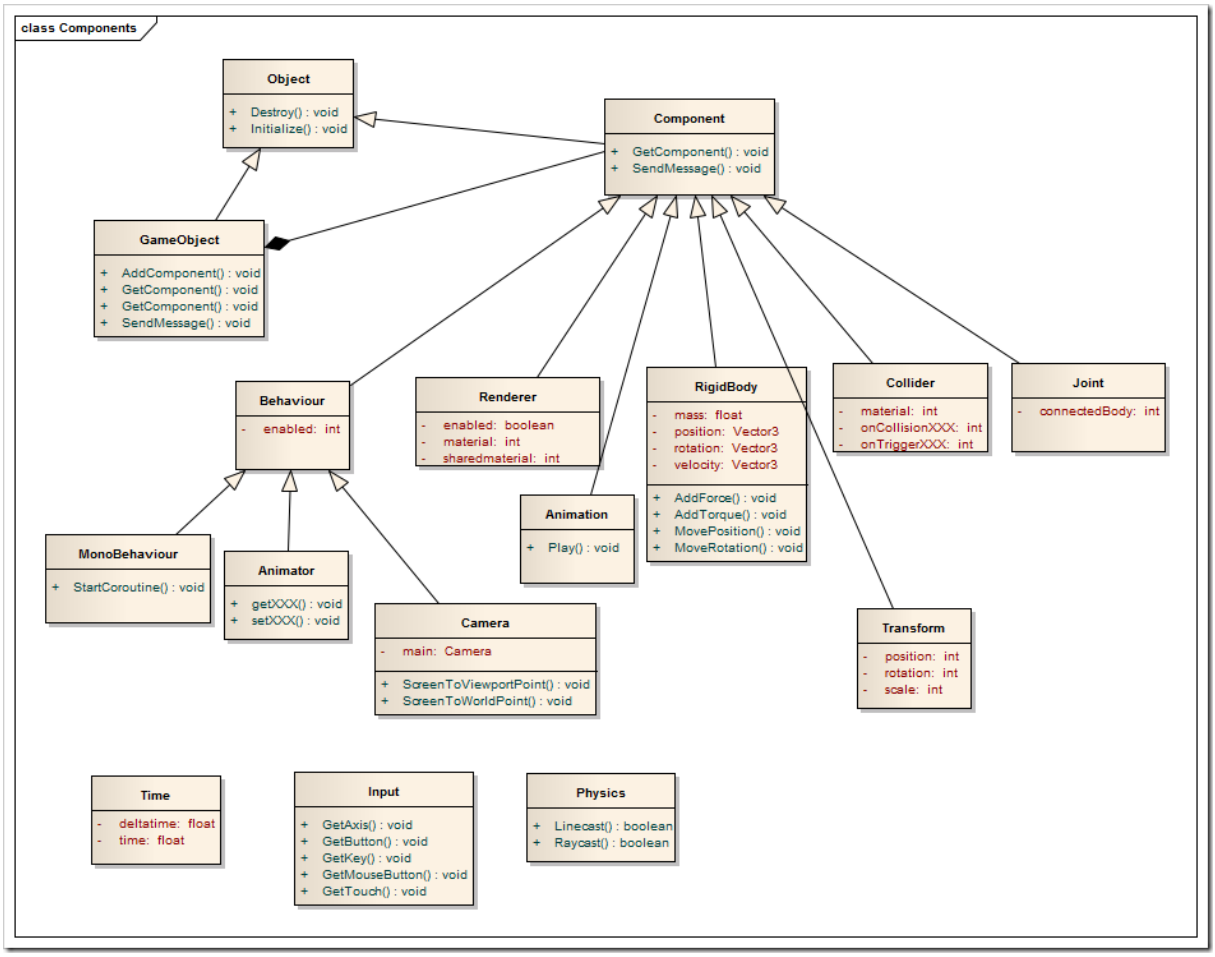


Figura 4.1: Diagrama de la arquitectura básica en Unity5

Por lo tanto, las clases que establecen el comportamiento de las unidades, las casillas, así como cualquier elemento del juego que ofrezca al usuario una visualización en pantalla, están implementadas en base a esta arquitectura, y son componentes de tipo *MonoBehaviour*. Para la comprensión de la implementación del proyecto es necesario decir que estas clases, debido a su condición de componentes, y a su herencia desde *MonoBehaviour*, tiene las siguientes características:

- No poseen constructor, son dependiente de su objeto físico.
  Las clases que heredan de *MonoBehaviour*, no pueden ser instanciadas *per se*, sino que se asignan como cualquier otro componente a un objeto de juego. De forma que al aparecer un objeto de juego, todas las clases *MonoBehaviour* que tenga definidas como componentes, son instanciadas internamente, y se puede acceder a ellas a través del propio objeto de juego, que a nivel de código se comporta de manera transparente, sin ser otra cosa que una composición de los componentes hayamos definido que debe tener este objeto de juego. (La nomenclatura que usa *Unity5* para estos objetos es *GameObject*, como se ve en la figura 4.1, y es la que usaremos a partir de aquí).
- Proporciona una captura de eventos automatizada.

Cualquier clase que herede de MonoBehaviour, podrá acceder de forma abstracta para el programador, a los eventos que sucedan sobre su *GameObject*, como pueden ser por ejemplo:

- Start()
  Método que es llamado cuando el objeto aparece en escena, esté habilitado o no.
- Awake()
  Método que es llamado cuando el objeto se habilita dentro de la escena.
- Update()
  Método que es llamado en cada nueva *renderización* del objeto, es decir, se va ejecutando continuamente cada vez que la escena se actualiza y se muestra la siguiente imagen en pantalla.

- Proporciona acceso a otros componentes complejos y permite manejarlos de forma sencilla.
  Un ejemplo de esto es tener un objeto de juego, sobre el que queremos realizar alguna acción cuando este entra en colisión con otro objeto, para ello simplemente debemos añadir al objeto un componente nativo de *Unity5* , *Collider*, que lanza un evento con todos los datos necesarios cada vez que detecta cualquier colisión o cualquier cambio en las colisiones que ya tuviera previamente. Desde una clase que programemos, que herede de *MonoBehaviour*, podemos capturar este evento e implementar lo que queramos que suceda según los datos recogidos por el componente *Collider*

4.1.2. Partículas y modelos 3D

Para ofrecer un entorno agradable, se ha utilizado la herramienta **Blender** para diseñar algunos elementos tridimensionales, como pueden ser las fichas de juego o las casillas, que tienen una leve profundidad. También se han añadido sistemas de partículas externos de la comunidad oficial de *Unity5* . Con los cuales se han podido crear ciertos efectos para ofrecer al jugador pistas visuales sutiles.

4.2. La interfaz y sus elementos

Es necesaria una tarea de diseño, en la que se determina qué es necesario visualizar, en qué momentos aparecen o desaparecen estos elementos, cómo se comportan respecto a la interacción con el usuario, y diseñarlos estéticamente.

4.2.1. Tablero y casillas

El primer elemento, y el que está presente durante todo el juego, es el tablero y las casillas que lo componen, el tablero no tiene una representación física, pero si las casillas. Una vez decidido en las reglas que estas casillas iban a ser hexagonales, se diseño el tamaño y fondo de las casillas, así como un modelado 3D que es el objeto que el usuario ve en pantalla. También

se diseñó una clase **CasillaFísica** que hereda de MonoBehaviour, la cual encapsula la lógica interna de la clase **Casilla**, en la figura 4.2, cada casilla que se ve es un *GameObject*, formado por el modelado y sus texturas, componentes predefinidos para detectar eventos con el usuario, y un componente del tipo **CasillaFisica**.

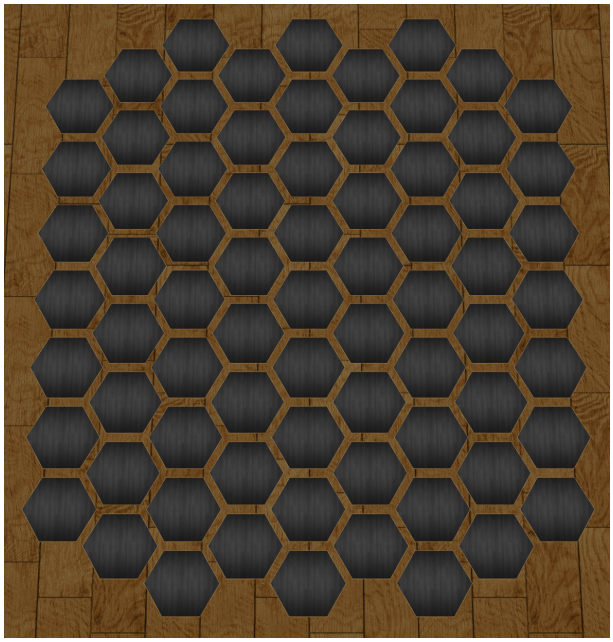
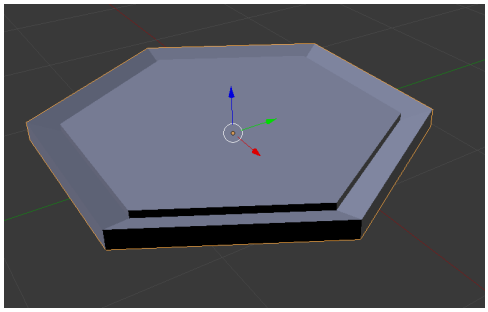


Figura 4.2: Tablero con sus casillas

4.2.2. Fichas de juego



(a) Objeto modelado

(b) Objeto manejado por *Unity5*

Figura 4.3: Representación visual de la ficha

A la hora de representar las unidades de juego, se plantearon varias formas, desde usar simples imágenes que sustituyeran la imagen por defecto de las casillas, hasta modelar personajes estáticos en 3D al estilo del ajedrez, los cuales se moverían por el tablero. Se optó por una solución intermedia, modelar una ficha de juego, la cual tendría también forma hexagonal, y el color del jugador dueño, y en el interior de la ficha la imagen de la unidad que representa. Primero se diseño el modelo en la herramienta **Blender**, importándolo posteriormente en **Unity5**, se pueden observar ambos editores en la figura 4.3

Para dotar a las unidades de comportamiento, se creo la clase **Ficha**, que hereda de *MonoBehaviour*, y por lo tanto la añadimos como componente a los *GameObject* que representan las unidades del jugador sobre el tablero.
El *GameObject* que representa una ficha, es el que se ve en la figura 4.3b, y tiene componentes de colisión, un componente de la clase mencionada **Ficha**, y componentes que permiten visualizar la imagen de la unidad y sus atributos de vida, defensa y ataque, que se ven en la parte inferior de la ficha de la figura 4.3b, siendo, en ese orden: 2 de vida, 2 de defensa y 1 de ataque.

4.2.3. Pistas visuales sobre el tablero

Para ayudar al jugador, teniendo en cuenta la experiencia de usuario, se han implementado además ciertas pistas visuales:

- Se han establecido transparencias de color para las casillas, aplicando un color blanco a las casillas sobre las que el usuario tiene el cursor.

- También se aplica un color verde a las casillas que componen la secuencia de un movimiento, de forma que el usuario ve el camino que va a seguir una unidad mientras elige dónde moverla. Para esto se usa el Pathfinder que calcula el camino entre dos **IGraphNode**, como se vio en la sección *Lógica interna*.

- Se ha creado un *GameObject* que gestiona el sistema de partículas previamente comentado, para establecer su comportamiento se ha creado una clase hija de *MonoBehaviour*, que se encarga de activar o desactivar la partícula necesaria, y de enlazarla con la **Casilla** o **Ficha** pertinente. Se ha denominado a esta clase **Runa**, por su aspecto. Cada **Casilla** o **Ficha** instanciará cuando aparezca en pantalla (con el método **Awake()**), su propia **Runa**, permitiendo así decirle a los elementos de juego que muestren una determinada partícula sobre ellos para mostrar información al jugador. En resumen, esta clase **Runa** es un gestor de partículas en el que se delega esta responsabilidad para no duplicar el código necesario para ello en ambas clases.

4.2.4. Modelado e implementación de los elementos visuales

Con los elementos visuales definidos, se modelan las clases necesarias para manejar estos objetos visuales, asociados a los elementos de la lógica interna:
Primero tenemos la **Runa** asociada a cada **Ficha** o **Casilla**, teniendo las runas una estructura de datos para acceder a los sistemas de partículas 3D, en forma de diccionario con claves, así, solo es necesario que cada elemento del juego llame a su runa activando o desactivando la clave correspondiente. Sin embargo será necesario, como se ve en puntos posteriores, un sistema para que estos elementos sepan qué activar o desactivar en cada momento, así como todas las interacciones con el usuario.
(Todos los métodos que manejan runas en los elementos de juego se denominan **MarcarComo...** y **DesmarcarComo...**, sustituyendo los "..."por el tipo de pista que está dando, si por

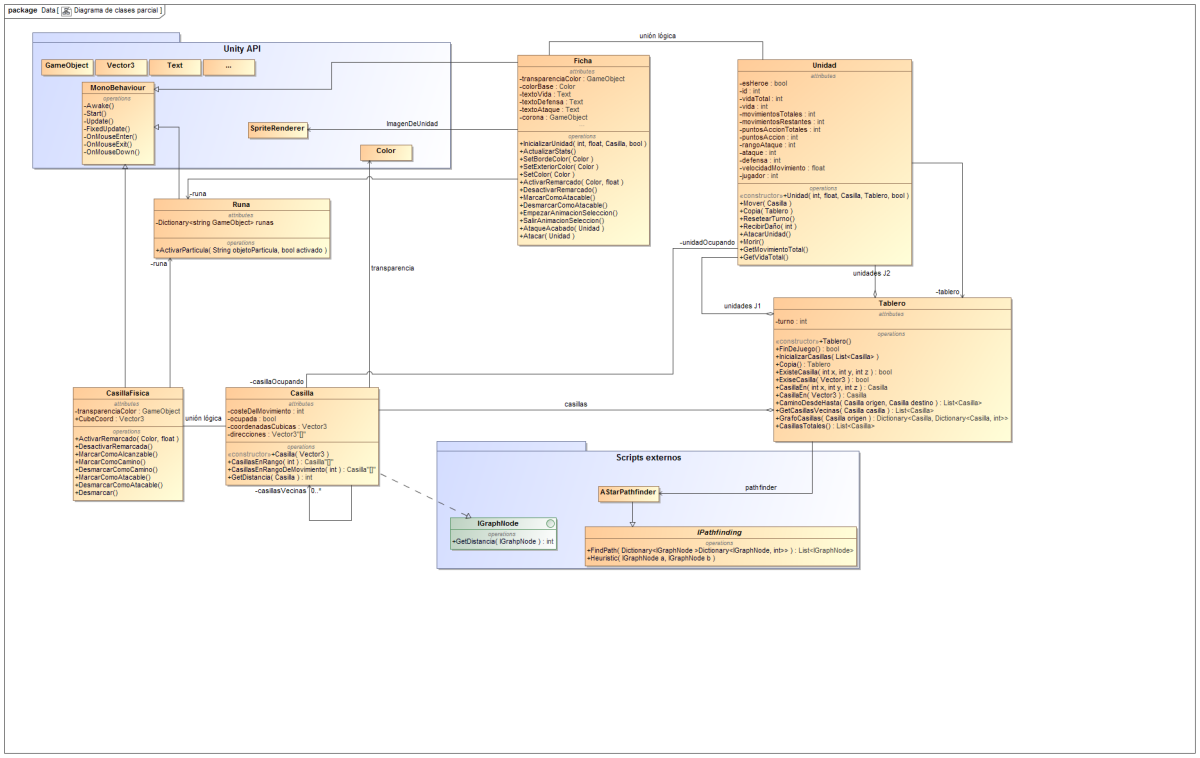


Figura 4.4: Diagrama inicial de elementos visuales

ejemplo es una casilla *alcanzable*, *atacable*, o forma parte del *camino*).

La clase **Ficha** ofrece otros elementos visuales, como puede ser una imagen de la unidad incrustada en la propia ficha, como se ve en la figura 4.3b, sobre la que se aplica un tinte para remarcar la unidad, dando información al usuario sobre el estado de la unidad en ese momento, también tiene elementos de Interfaz 2D, que se verán posteriormente, y el color de la ficha en sí, llamado *Borde* en el modelado, depende del jugador que tiene control sobre ella, dejando así claro al usuario sus unidades aliadas y enemigas, a través de colores complementarios. Los métodos de **Ficha** son los necesarios para poder manejar la visualización de esta, activando con un determinado color y transparencia el tinte sobre la imagen de la unidad, o desactivándolo, con los métodos **ActivarRemarcado** y **DesactivarRemarcado** respectivamente. Se puede establecer el color, o la runa, con otros métodos de la clase, de forma análoga, también se puede ver que se han modelado métodos para entrar y salir de una animación de selección, para ofrecer al usuario un *feedback* sobre cuál de sus unidades es la que tiene seleccionada, en caso de tener alguna.

La última clase a terminar de comentar, **CasillaFísica**, consiste simplemente en una serie de métodos para activar o desactivar el remarcado gracias al tintado de la casilla, igual que en la **Ficha**, y para activar o desactivar runas.

Una vez que se han implementado dichas funciones, es posible una primera visión del juego, incluyendo esto pistas visuales reales calculadas a través de la lógica interna. La figura 4.5 es

un ejemplo de algunas.

- Runas blancas
  Definen las casillas a las que se puede mover una unidad, en una sola acción de movimiento. Se calculan en ultima instancia gracias al método **GetCasillasEnRangoDeMovimiento(int rango)**, de la clase **Casilla**, el cual es llamado por la unidad, con su rango de movimiento como argumento.

- Casillas con transparencia verde
  Cuando se sitúa el cursor en una casilla a la que la unidad se puede mover, se tintan de verde las casillas del camino necesario para llegar a ella, siendo este tinte más opaco en la casilla donde quedaría situada al final la unidad, ya que es esta la casilla importante para el jugador en cuanto a estrategia de juego se refiere. Esta secuencia de casillas se obtiene a través del algoritmo A* ya comentado en la sección *Lógica interna*.

- Runas rojas pequeñas
  Cuando no se tiene el cursor sobre ninguna casilla transitable, se muestra el rango de ataque de la unidad, y en un caso como este, donde se tiene el cursor situado sobre una casilla a la que la unidad se puede mover, estas runas muestran donde podría atacar la unidad en el siguiente turno desde ahí. Se obtiene a través del método **GetCasillasEnRango(int rangoAtaque)**, de la clase **Casilla**, el cual es llamado por la unidad, con su rango de ataque como argumento.

- Runas rojas y azules grandes
  Funcionan como las rojas pequeñas anteriormente mencionadas, sólo que tienen este tamaño y color más grande en el caso de que sobre la casilla haya una unidad enemiga, para resaltar una unidad a la que se puede atacar. Si existe una unidad aliada sobre una casilla atacable, no se muestra con ninguna runa, dejando claro al jugador que no puede realizar ninguna acción sobre ella.

(a) Estado de juego inicial

(b) Ficha seleccionada

(c) Pistas del movimiento hasta la casilla realzada

Figura 4.5: Pistas visuales para la interacción

5. Interacción con el usuario

5.1. La clase Juego

Una vez implementados parcialmente los elementos de juego, ya tenemos una primera idea de lo que va a ver el usuario en pantalla, pero es necesario implementar la interacción persona-ordenador, es decir, como el usuario interacciona con el juego,tanto captando la información como realizando acciones sobre éste.

Unity5 proporciona a los *GameObject* que tienen un componente de tipo *collider*[9] funciones para captar los clics del usuario. Esto se hace creando un *GameObject* lineal invisible al usuario llamado *raycast*, cuya dirección es la que forma el vector cuyo origen es la posición de la cámara de juego, que es la que renderiza lo que ve el usuario en su pantalla, y el fin del vector es el punto de la pantalla en el que el usuario tiene el puntero, como se ve en la figura 5.1

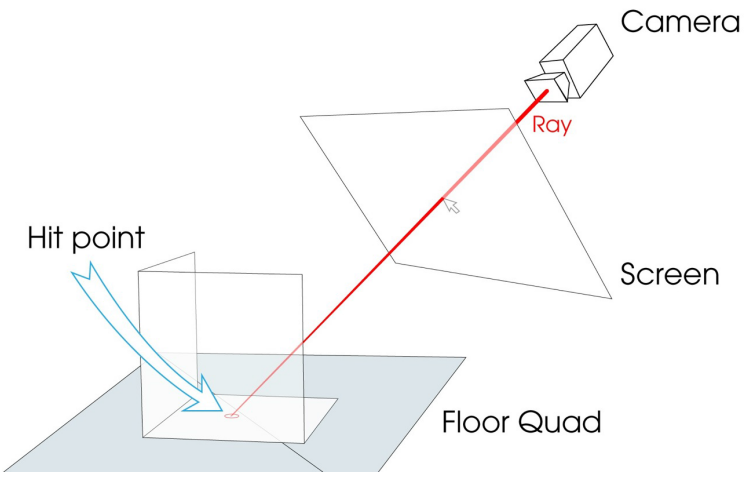


Figura 5.1: Ejemplo de clicar sobre el objeto suelo, (*GameObject* Floor Quad)

Este vector se prolonga infinitamente dentro de la escena de juego, devolviendo los datos de colisión, (el *GameObject* con el que ha colisionado, el punto en el que esto se está produciendo, etc.), a las funciones que esperen una llamada de este evento. En caso de que no se produzca ninguna colisión, o no haya ninguna función esperando este evento, no se produce

ninguna llamada y no es necesario manejar dicha falta de colisión.

Podemos por lo tanto, tener en los elementos gráficos del juego interaccionables, tanto fichas como casillas, diferentes eventos capturando diferentes acciones del ratón sobre ellos:

- Hacer clic
- Pasar de no tener el puntero sobre el objeto a sí tenerlo en un determinado *frame*
- Pasar de tener el puntero sobre el objeto a no tenerlo en un determinado *frame*
- Mantener el puntero sobre el objeto
- Hacer clic y arrastrar o mantener pulsado

También podemos detectar las interacciones del usuario con el teclado, obteniendo las teclas que pulsa, mantiene o suelta, de la misma forma.

Una vez tenemos implementada la detección de los eventos del usuario es necesario manejarla, nos encontramos ante el problema del manejo de forma global, hacer clic sobre un elemento no tiene por qué tener que afectar solo a ese elemento. Para ello se añade una clase que es clave para el desarrollo de los elementos visuales dentro del juego, la clase **Juego**. Hereda de *MonoBehaviour* como un mero trámite para ser instanciada al inicio del juego, y ser capaz de detectar acciones del usuario como pueden ser pulsaciones del teclado.

Al definir esta clase y añadirla al modelo global, se obtiene una base para el juego, ya que la clase **Juego** tiene toda la información necesaria para actuar sobre los elementos visuales del juego, y estos a su vez sobre su lógica interna acoplada, quedando así el diagrama de clases: Como se ve en el diagrama de la figura 5.2, se asocia al **Juego** un tablero, a través del cual accede a todos los elementos del juego, así se podrá centralizar el comportamiento general del juego en esta clase, ofreciendo métodos de captura de eventos y acciones, y ejecutando las respuestas pertinentes sobre estos y sus representaciones físicas, es decir, se efectúa una vía de comunicación de doble sentido en la que el usuario realiza una acción, esta acción es captada por el componente *Collider* del elemento en cuestión, y lo transmite al juego, que decide que hacer con dicha acción, generando, en caso de que sea necesario, una respuesta al tablero, cambiando así alguno de los elementos lógicos de juego, de lo cual tienen constancia sus representaciones físicas, ofreciendo así al usuario una nueva visualización del estado de juego consistente con el estado de juego lógico.

Las funciones de **Juego** son las que captan las siguientes interacciones del usuario sobre **fichas** y **casillas**:

- **CasillaRealzada(CasillaFisica casilla)** y **FichaRealzada(Ficha ficha)**
  Cuando se sostiene el puntero sobre el elemento.

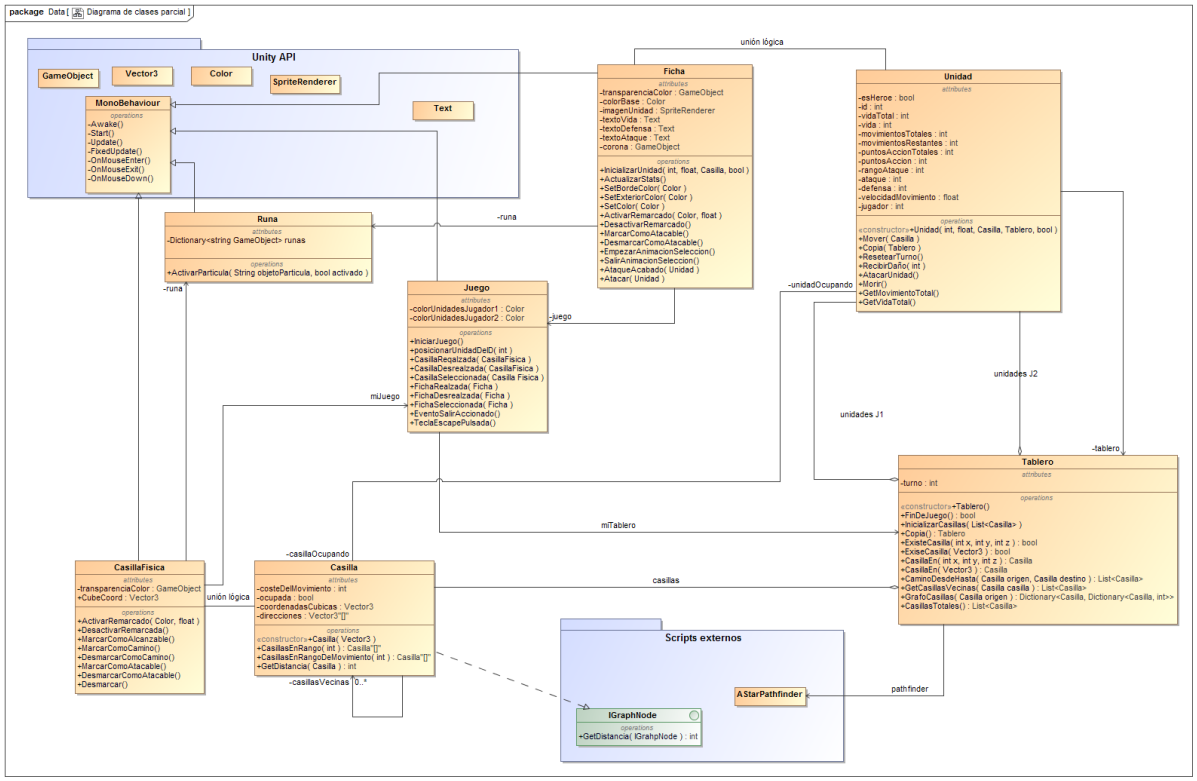
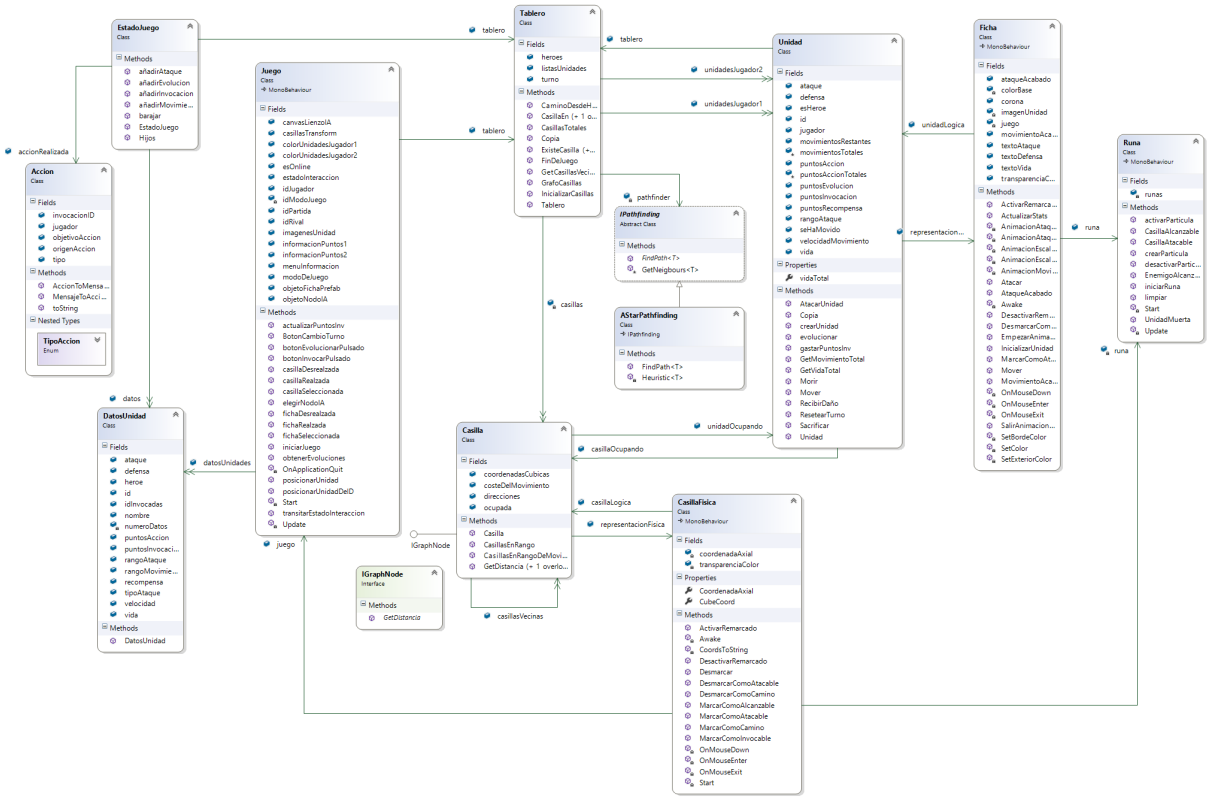


Figura 5.2: Diagrama de clases centralizado en la clase Juego

- **CasillaDesrealzada(CasillaFisica casilla)** y **FichaDesrealzada(Ficha ficha)**
  Cuando el puntero sale trás haber estado realzando el elemento.

- **CasillaSeleccionada(CasillaFisica casilla)** y **FichaSeleccionada(Ficha ficha)**
  Cuando se detecta un clic(izquierdo, ya que usamos el *estándar de facto* de usar este clic para la selección) sobre el elemento.

También captará eventos que se manejarán indistintamente de la posición puntero, como puede ser detectar pulsaciones de teclado o el clic derecho del ratón. Una vez definidas estas clases, y diseñados estéticamente los elementos gráficos, se procede a la implementación de ellos, difiriendo bastante la implementación final del modelado inicialmente planteado, esto es debido a que la arquitectura de *Unity5* es más compleja de lo planteado, y se necesitan implementaciones más trabajadas para cumplir los requisitos planteados, el diagrama final, extraído directamente de la implementación es el siguiente:
(Nota: Al tratarse de un diagrama de la implementación final, existen campos y funciones que hacen referencia a siguientes capítulos de la memoria).



Índice de figurashola

Figura 5.3: Diagrama la implementación final de los elementos gráficos del juego, con la lógica interna acoplada a estos

Siendo algunos elementos nuevos los siguientes:
La clase **DatosUnidad** que almacenan todo lo relativo a la naturaleza de las unidades y sus datos estáticos, siendo esta clase una representación de los datos que recogemos de la base de datos, los cuales se traducen a un objeto de tipo **DatosUnidad** al cargarlos, para evitar llamadas innecesarias al servidor.
Como se puede observar, todas las clases tienen una complejidad mayor de la inicialmente planteada, algunos de los métodos imprevistos son muy sencillos, pero otros han requerido de un gran esfuerzo para su implementación. Conseguimos con esta implementación final una base ya programada para el videojuego, que además consta de una gran flexibilidad, ya que se debe tener en cuenta que para el trabajo se ha considerado el producto mínimo viable, pero se ha implementado una arquitectura base que permita añadir futuras mecánicas de juego muy diferentes. Por ello, por ejemplo este esqueleto se abstrae de la posición de las casillas y su estructura, de las acciones de las unidades de un jugador, de la definición de acciones posibles en un turno, etc.

5.2. Implementación de la interacción real

Se podría decir que en este punto nuestro producto ya es usable, sin embargo la implementación de la figura 5.3 carece de varias funciones básicas, una de ellas es la resolución de que hacer al detectar una interacción del usuario. Es necesario restringir lo que el usuario puede hacer según el estado de juego, por ejemplo, sucederá algo distinto si el usuario selecciona una ficha, si es su turno o si no. Por ello, según el estado del juego, se deberán permitir acciones diferentes y ofrecer pistas visuales diferentes, que concuerden con las acciones realizables. Y también será necesario ofrecerle al usuario una respuesta que pueda procesar sobre sus acciones, por ejemplo si un usuario ataca con una unidad a otra, no podemos simplemente disminuir la vida de la unidad enemiga, o si el usuario mueve una unidad, simplemente que aparezca al instante en la casilla destino. Para estos dos fines, la restricción de los usuarios y sus respectivas respuestas, se ha establecido una máquina de estados, la cual se ha usado en el juego para establecer en qué momento de interacción con el juego se encuentra el usuario, y se ha implementado a través del **patrón de delegación**

5.2.1. Estado de Interacción

Primero es necesario definir en que estados se va a poder encontrar el juego de cara al jugador, nos encontramos con los siguientes:

- Inicio del juego
  Se trata del estado en el que aún se están realizando procesos iniciales, (carga de datos, carga de modelos y texturas, informar al usuario de qué jugador empieza, etc.). En este estado el jugador no puede seleccionar fichas.

- Esperando acción
  En este estado se entra cuando es el turno del propio jugador, y por lo tanto se le permite seleccionar sus unidades. También se le ofrecen pistas al realzar sus unidades antes de seleccionar alguna de ellas.

- Ficha seleccionada
  Cuando un jugador selecciona una ficha, pasa a tenerla seleccionada y se ofrecen las pistas visuales de lo que puede hacer con ella, además, se permite seleccionar unidades a las que puede atacar con ella, o casillas a la que se puede mover. También permite evolucionar a la unidad o invocar una nueva, dependiendo del tipo de unidad y de si tiene puntos para ello. Cuando seleccione una de estas acciones se ejecutará la acción sobre el tablero y se mostrará el resultado visual al usuario.

- Invocando
  Cuando el jugador elige invocar una unidad, se muestra dónde puede situarla y se le permite seleccionar una de esas casillas.

- Realizando acción
  Cuando se está ejecutando alguna acción que lleva asociado un tiempo de transición,

como puede ser el caso de las animaciones de ataque o movimiento, se usa este estado para bloquear ciertas acciones del usuario hasta que termine la acción.

- CambioDeTurno
  Este estado es en el que se realiza el cambio del turno, y mientras esto sucede ningún jugador puede realizar acciones.

Dando lugar esta especificación a la siguiente máquina de estados (figura 5.4).

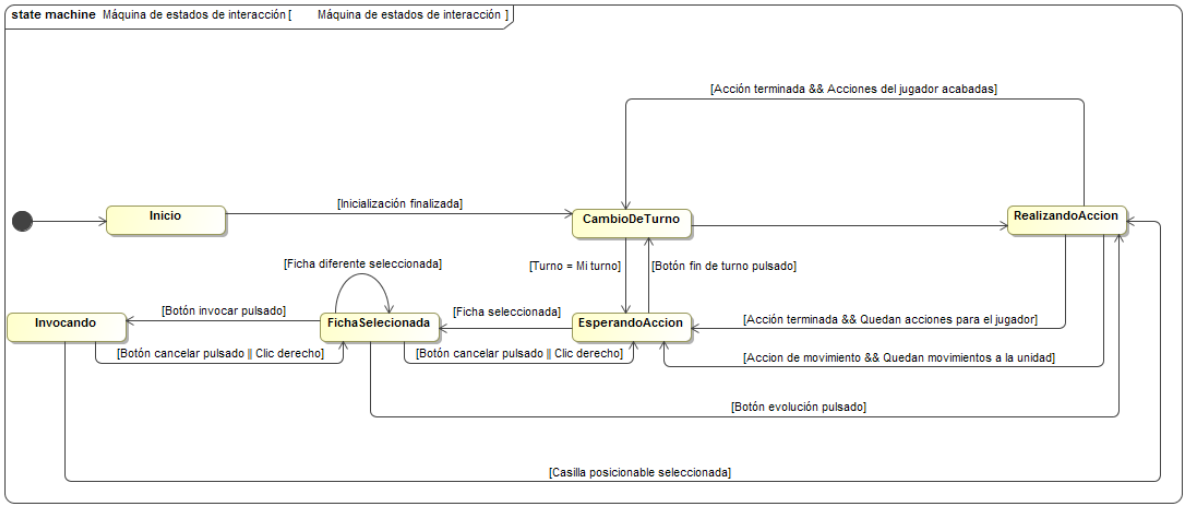


Figura 5.4: Máquina de estados de interacción

5.2.2. Delegación sobre el estado de la interacción

Ya que las funcionalidades de interacción con el usuario recaían en la clase **Juego**, pero esta no tiene conocimiento de en qué estado de los mencionados se encuentra, se ha implementado un **patrón delegación**, por el cual existe una clase abstracta que recoge los eventos sucedidos, y las respuestas a estos se sobreescriben en la clase que representa a cada estado, el modelado consiste en la clase abstracta **EIBase**, la cual determina una serie de funciones comunes a todos los estados, y además implementa en ellas una funcionalidad base que todos los estados cumplen. (Diagrama clases de la figura 5.5)
Nótese que tiene además, las funciones **EntradaAlEstado()** y **SalidaDelEstado()**, los cuales se ejecutan en dichos momentos para cada estado.

La clase **Juego** tiene, por lo tanto, un objeto de tipo **EIBase** asociado en el cual delega las respuestas a las interacciones del usuario, abstrayéndose así de cual es la clase instanciada en ese momento. Por otra parte, el objeto EIBase está encapsulado en un campo privado, a través del cual sólo es posible cambiar la clase instanciada a través del método **transitarEstadoInteraccion(EIBase nuevoEstado)**, ejecutando el método de salida del estado, cambiando el objeto instanciado, y ejecutando el método de entrada al estado del nuevo objeto. Para el usuario existe, por tanto, un *feedback* estándar, que es el que se implementa en la

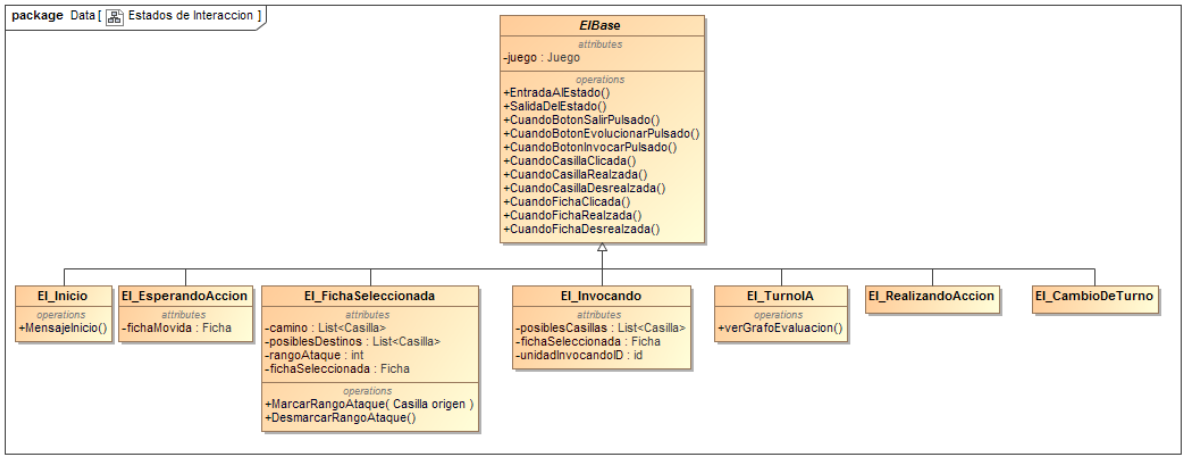


Figura 5.5: Diagrama de clases los estados de interacción

clase abstracta, y luego cada acción establece diferentes formas de interaccionar, por ejemplo, el estado **FichaSeleccionada**, será en el que se puede dar una situacion visual como la de la figura 4.5. Además este estado actúa a modo de observador, cambiando las runas que marcan las zonas atacables según el jugador mueve el ratón por las casillas.

5.2.3. Reacción visual

Una vez permitido que el jugador actúe de forma legítima sobre el tablero, se determinan una serie de respuestas visuales sobre las acciones del jugador, aparte de las pistas visuales ya implementadas, en este caso hablamos de animaciones de las fichas. Son necesarias tres, que luego se subdividen según el tipo de unidad.

- Movimiento La ficha se moverá suavemente entre las casillas que forman el camino ya calculado.
- Ataque La ficha chocará contra el objetivo, o lanzará un proyectil, dependiendo del tipo de ataque y su rango.
- Selección Al tener una ficha seleccionada se establecerá un zoom periódico sobre la ficha para ofrecer al usuario un *feedback* sobre su selección.

Todas estas animaciones se ejecutan en subrutinas en segundo plano, un ejemplo es la animación de movimiento.

```
IEnumerator AnimacionMovimiento(List<Casilla> camino)
{
  camino.Reverse();
  foreach (Casilla c in camino)
  {
    while (new Vector2(transform.position.x, transform.position.y) != new
    Vector2(c.representacionFisica.transform.position.x, c.
    representacionFisica.transform.position.y))
    {
```

```
        transform.position = Vector3.MoveTowards(transform.position, new
    Vector3(c.representacionFisica.transform.position.x, c.
    representacionFisica.transform.position.y, transform.position.z), Time.
    fixedDeltaTime * unidadLogica.velocidadMovimiento);
9       yield return 0;
     }
11  }
    if (unidadLogica.movimientosRestantes != 0)
13  {
     juego.transitarEstadoInteraccion(new EI_FichaSeleccionada(juego, this))
    ;
15  }
    MovimientoAcabado();
17 }
```

En este código se puede ver como, primero se obtiene la lista inversa de casillas, para obtener el camino desde el punto de vista de la ficha, y luego para cada una de las casillas, hasta que la ficha esté en la casilla final, se va moviendo la ficha desde su posición a la de la siguiente casilla del camino gradualmente teniendo en cuenta el tiempo real que pasa entre dos ejecuciones de dicho método con el parámetro *Time.fixedDeltaTime*, cuando el movimiento termina, si quedan movimientos pasamos directamente al estado de juego en el que tenemos la ficha seleccionada pero con sus movimientos restantes reducidos los que se hayan gastado en dicho movimiento.
Si no quedan movimientos restantes, el jugador no tiene más posibles acciones según nuestras reglas actuales, y por ello el turno acaba automáticamente.
El resto de animaciones se implementan de forma análoga.

5.3. Modos de juego

En este proyecto se han definido varios modos de juego: Contra la Inteligencia Artificial, contra otro jugador en línea o contra otro jugador de forma local. Vista la efectividad de estructurar diferentes posibilidades a través de un **patrón de delegación**, se ha vuelto a implementar una delegación de la clase juego sobre una clase abstracta, no nos centraremos mucho en ello ya que se ha implementado de forma análoga a la delegación previamente mencionada, pero es importante decir que, mientras que la delegación sobre el estado de interacción se basaba en los eventos, esta nueva delegación sobre el modo de juego se basa en la realización de acciones, sobretodo una vez finalizadas, ya que el modo de juego debe decidir si enviar ciertos datos al servidor y esperar la respuesta, o no manejar ningún elemento en red, o decidir si ejecutar agentes inteligentes o no, etc.

Si añadimos estas dos delegaciones al diagrama de la figura 5.3, tenemos el diagrama final de la implementación. Las extensiones a dicho diagrama serían las siguientes:

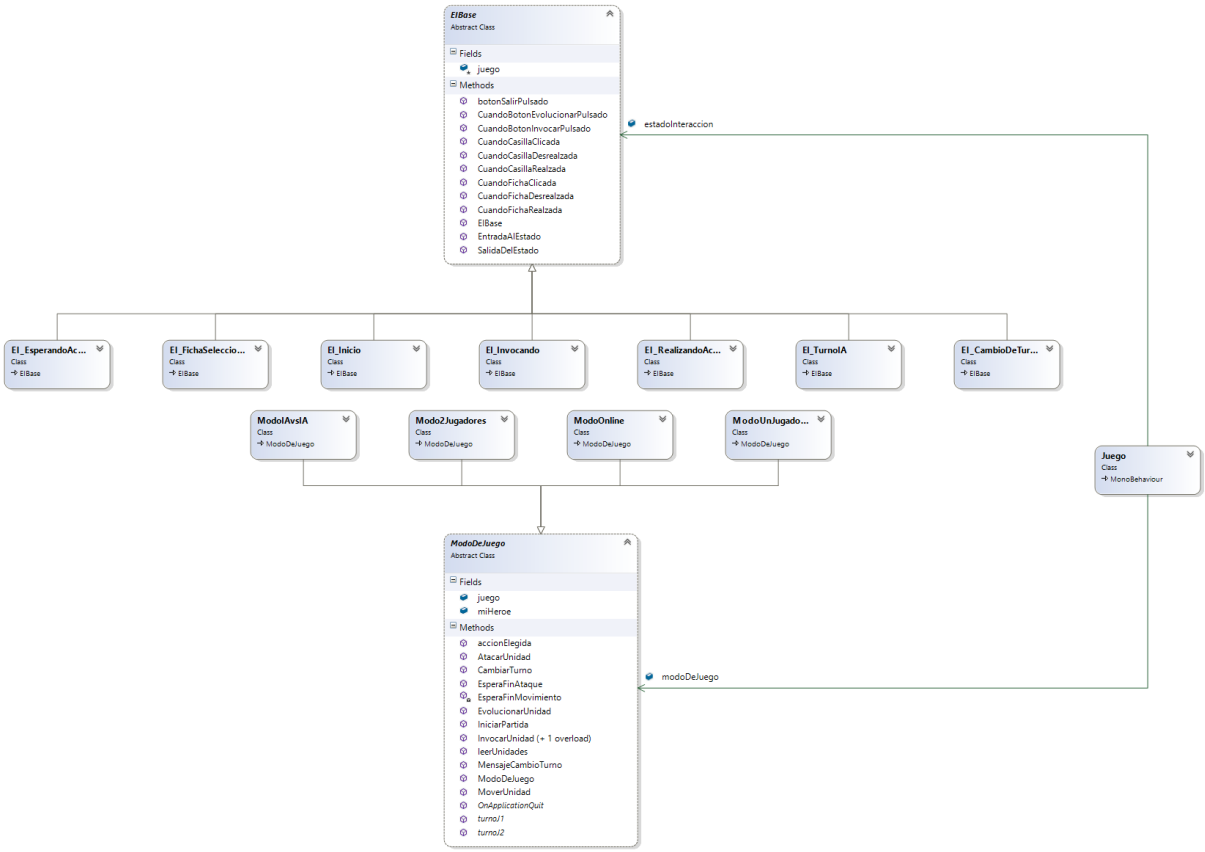


Figura 5.6: Extensión de la clase Juego con sus delegaciones de funcionalidad

6. Inteligencia Artificial

Una vez el usuario puede realizar acciones sobre el juego, podemos establecer un sistema para que dos jugadores se alternen para jugar en un mismo ordenador, sin embargo es necesario ofrecer otro tipo de oponentes para el usuario.
Se ha definido una interfaz **ElectorDeAccion**, que establece la función **GetAccionElegida**, la cual devuelve una acción, en el caso del modo de juego en línea, esta acción la recibiremos del servidor, pero en el caso del modo de un jugador, es necesario implementar un algoritmo que nos devuelva una acción, la cual sea posible sin incumplir las reglas de juego.
En el proceso de implementación de dicho algoritmo se han elaborado diferentes estrategias de búsqueda, las cuales se explican a continuación.

6.1. Búsqueda simple de estado adyacente

La primera, y más básica de las estrategias, es la búsqueda simple de estados adyacentes, implementada en la clase **BusquedaDeAccionSimple**, la cual obtiene un **EstadoDeJuego** y un **Evaluador**. Se basa en generar los hijos del estado de juego recibido, obtener la evaluación de cada uno de ellos, usando el evaluador recibido, y devolver la mejor acción posible, que es la que lleva del estado de juego recibido, a aquel de sus hijos con mayor evaluación.

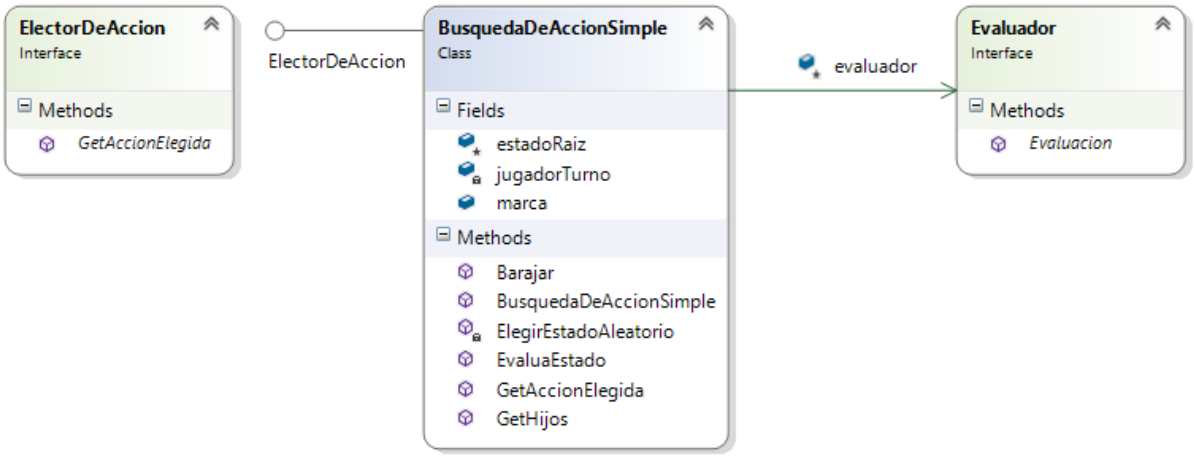


Figura 6.1: Implementación de la búsqueda simple

Estrategia de evaluación

Como se puede observar en la figura 6.1, existe un método *EvaluaEstado*, que recibe un objeto **Evaluador**, que proporciona un método *Evaluación* destinado a obtener una puntuación para cada estado que se envíe como parámetro. Esta estructura para delegar la estrategia de evaluación en otro objeto se ha hecho con la finalidad de diferenciar la estrategia de búsqueda de estados y la estrategia de evaluación de dichos estados. Es la unión de un algoritmo de búsqueda específico, y una forma de evaluación, la que da como resultado un *jugador enemigo*

Se han probado varias estrategias de evaluación, y dada la inmensidad de las posibles situaciones del juego, cuanto mejor funcionaba una evaluación en una determinada situación, peores resultados daba en otras situaciones diferentes. El evaluador con el que se han tenido mejores resultados es el **Evaluador Parametrizado**, que se creó con la intención de aglutinar las tres estrategias de evaluación que, al probarlas, resultaron más interesantes, debido a ello consiste en generar tres puntuaciones distintas (una por cada estrategia incorporada), proporcionando una forma de cambiar la importancia que se le da a cada puntuación, en forma de parámetros, además, ofrece otros parámetros que cambian como se genera internamente cada puntuación. Las tres puntuaciones son las siguientes:

- Salud:
  Obtiene una puntuación teniendo en cuenta únicamente el número de unidades, su vida y su defensa. También se incluye dentro de esta puntuación los puntos de invocación o de evolución.

- Ataques:
  Obtiene una evaluación teniendo en cuenta los posibles ataques que se podrían realizar una vez en ese estado, además, cuanto más potente sea el ataque, y cuanto más importante sea la unidad atacada, mayor valor tiene.

- Posición:
  Obtiene una evaluación teniendo en cuenta las posiciones de las unidades, esta evaluación es la más abstracta, ya que genera un valor a partir de la distancia entre unidades, asumiendo que una unidad con poca salud deberá huir mientras que unidades con mayor salud deberán atacar, sin embargo, dependiendo del momento de juego, un movimiento de sacrifico de una unidad es algo mucho más positivo que una huida. Sin embargo, se calcula este parámetro así para poder obtener una evaluación primitiva para el producto mínimo viable, ya que nuestro juego tiene una gran cantidad de posibilidades y de diferentes estrategias, y tras comparar varias, esta evaluación de la posición de las unidades es la más estable en cuanto a ventajas y desventajas. Cualquier intento de implementación de mecánicas de sacrificio de unidades ha desembocado en jugadores demasiado agresivos. Y al trabajar para mejorar las evaluaciones han resultado en funciones de evaluación tan complejas que lastraban la ejecución del algoritmo de búsqueda de estados.

Los parámetros recibidos permiten modificar cómo se genera la puntuación, por ejemplo la importancia que se le da a la vida frente a la defensa, así como al héroe frente al resto de

unidades. Y una vez generada cada puntuación, se obtiene una evaluación total mediante la ponderación de las tres puntuaciones.

Esta ponderación altera la "personalidad"del jugador, ya que por ejemplo, al dar un valor mayor a la evaluación de los ataques genera jugadores más cautos, que no realizan acciones a no ser que le den más posibilidades que al enemigo, mientras que dar un mayor valor a la salud genera jugadores más reactivos", que se atreven a ejecutar un ataque, aunque les pueda poner en peligro en el futuro, y que huyen de un ataque próximo, incluso si con ello dan al enemigo una posibilidad de intentar otro ataque.

Cálculo de los estados hijos

Se han usado funciones de **C#** más complejas que para el resto de la implementación, ejecutando las búsqueda de hijos en hilos, sobre los cuales se maneja su prioridad y su evento de finalización, para poder proseguir con los siguientes hijos. El resultado es complejo, ya que el factor de ramificación de un estado normal de juego puede ser por ejemplo:
7 unidades x 20 movimientos de media = 140 acciones de movimiento.
7 unidades x 2 unidades enemigas al alcance = 14 acciones de ataque.
3 acciones de invocación inicial.
7 unidades x 3 posibles evoluciones = 21 acciones de evolución

Lo que significa 178 posibles estados hijos, pero en algunos casos de prueba se ha llegado a más de 200 hijos e incluso 300 en casos extremos, donde las 7 unidades aliadas sobre el tablero tenían 3 casillas tanto de rango de movimiento como de ataque. Si tenemos en cuenta que en el ajedrez, el factor de ramificación medio es de entre 35 y 38 posibilidades, se puede ver por qué es importante no generar una función de evaluación demasiado compleja, y además el cálculo de hijos, pese a ser ejecutado muy eficientemente, tiene un tiempo de ejecución bastante largo.
Además, mientras que en este tipo de juegos, como el ajedrez, a medida que el juego avanza, se pierden unidades, en el juego que hemos desarrollado se ganan, y estas a su vez mejoran sus rangos de movimiento y ataque, lo que hace que a medida que es más necesario generar estrategias más interesantes, es cuando el algoritmo es más pesado.

6.2. Búsqueda predictiva de estados

Esta nueva estrategia extiende la clase **BusquedaDeAccionSimple**, basándose en la misma obtención de hijos y el mismo método de evaluación de esta.
La intención de esta nueva estrategia es tener en cuenta las acciones futuras, es decir, evaluar el árbol de estados que se obtiene al generar todos los posibles hijos de cada estado hasta una cierta profundidad. Durante el proceso de investigación para implementar esta estrategia, se tuvieron en cuenta varios algoritmos clásicos de la Inteligencia Artificial, pero ninguno era por sí mismo una solución. El algoritmo A* necesita una heurística, la heurística para nuestro juego

podría ser el número de turnos necesario para derrotar al héroe enemigo, asumiendo que el enemigo va a permitir que lo hagamos de la forma más eficiente. Sin embargo esta heurística dista mucho de ser buena, y además, no se tiene en cuenta la resistencia del enemigo.
Otro algoritmo valorado es el MiniMax, sin embargo, en nuestro juego, desde cualquier estado se puede generar un árbol infinito, ya que las acciones de movimiento son viables en cualquier estado, y no generan ninguna ventaja por si mismas, sino que permiten acercarnos para realizar ataques o alejarnos para evitarlos. Por lo que no es viable un algoritmo basado en obtener valoraciones de los estados finales.

Por ello, el algoritmo realizado no es ninguno de los mencionados, sino una mezcla de los conceptos de estos y otros algoritmos: Se ha implementado una búsqueda recursiva, acomodando los principios del algoritmo **NegaMax** a nuestras necesidades, estableciendo una profundidad máxima,de forma que en lugar de parar en los estados finales, paramos en los estados a profundidad máxima.

```
1  public NodoBusqueda BusquedaNegada(EstadoJuego e, int profundidad, int
       jugador){
     EstadoJuego mejorEstado = null;
3    float mejorEvaluacion = Mathf.NegativeInfinity;
     float nuevaEvaluacion;
5    if (profundidad == 0 ){
       mejorEvaluacion = jugador * EvaluaEstado(e);
7    }else{
       foreach (EstadoJuego e2 in e.Hijos(new MarcadorTiempo())){
9        nuevaEvaluacion = -(BusquedaNegada(e2, profundidad - 1, -jugador).
       evaluacion);
         if ((nuevaEvaluacion > mejorEvaluacion) || (mejorEstado == null)){
11         mejorEvaluacion = nuevaEvaluacion;
           mejorEstado = e2;
13       }
       }
15   }
     return new NodoBusqueda(mejorEstado, mejorEvaluacion);
17 }
```

6.3. Búsqueda predictiva de estados con poda

Esta estrategia es el resultado de aplicar más conceptos de algoritmos conocidos, sustituyendo el algoritmo **BusquedaNegada** visto anteriormente por dos métodos distintos que se encargan de adaptar la poda alfa-beta a nuestras necesidades. Son dos algoritmos que intentan encontrar el mejor estado para el jugador, sea el usuario o el oponente, de forma que vamos oscilando entre los algoritmos al ir cambiando la profundidad de búsqueda (y con ello el turno del jugador), y, de igual forma que en el anterior algoritmo, se establece una profundidad máxima, por lo que tomaremos como nodos hoja aquellos en los que un jugador ha ganado, o los que se encuentran a profundidad máxima.

```
NodoBusqueda BusquedaMejorEstadoOponente(EstadoJuego e, NodoBusqueda alpha,
    NodoBusqueda beta, int profundidadRestante){
  if (profundidadRestante == 0 || alpha.evaluacion == Mathf.Infinity ||
   beta.evaluacion == Mathf.NegativeInfinity) {
    NodoBusqueda nuevo = new NodoBusqueda(e, evalua(e));
    return nuevo;
  }else{
    foreach (EstadoJuego h in e.Hijos(marca)){
      NodoBusqueda posibleMin = alphaBetaMin(h,alpha, beta,
   profundidadRestante - 1);
      NodoBusqueda maxElegido = new NodoBusqueda(h,posibleMin.evaluacion);
      float puntuacion = posibleMin.evaluacion;
      if (puntuacion >= beta.evaluacion){
        return beta;}   // corte beta
      if (puntuacion > alpha.evaluacion){
        alpha = maxElegido;} // alfa actua como max
    }
  return alpha;
  }
}

NodoBusqueda BusquedaMejorEstadoUsuario(EstadoJuego e, NodoBusqueda alpha,
   NodoBusqueda beta, int profundidadRestante){
  if (profundidadRestante == 0 || alpha.evaluacion == Mathf.Infinity ||
   beta.evaluacion == Mathf.NegativeInfinity) {
    NodoBusqueda nuevo = new NodoBusqueda(e, evalua(e));
    return nuevo;
  }else{
    foreach (EstadoJuego h in e.Hijos(marca)){
      NodoBusqueda posibleMax = alphaBetaMax(h, alpha, beta,
   profundidadRestante - 1);
      NodoBusqueda minElegido = new NodoBusqueda(h, posibleMax.evaluacion);
      float puntuacion = posibleMax.evaluacion;
      if (puntuacion <= alpha.evaluacion){
      return alpha;}   // corte alfa
      if (puntuacion < beta.evaluacion){
      beta = minElegido;} // beta actua como min
    }
  return beta;
  }
}
```

Este jugador tiene un gran problema, al establecer una profundidad determinada, puede que dependiendo del ordenador que ejecute el juego tarde más o menos el algoritmo, de forma que mientras en un ordenador de gama media-alta de sobremesa puede dar un buen resultado, en un portátil de gama baja puede tener la consecuencia de que el juego sea injugable. Este problema es el que se intenta corregir con la siguiente implementación.

6.4. Búsqueda con límite de tiempo

Para solucionar el problema de la anterior estrategia, se añade a la implementación anterior un temporizador, de forma que se puede establecer no sólo una profundidad máxima, sino también un tiempo de ejecución máxima.
Sin embargo, presenta el problema contrario, en un ordenador muy lento, esta vez no será injugable por tiempo, pero la búsqueda será tan lenta que el enemigo artificial jugará de forma mucho peor. Por esto, ahora depende del enfoque que se le quiera dar al producto, elegir un algoritmo u otro.

6.5. Diagrama final de las estrategias y su implementación

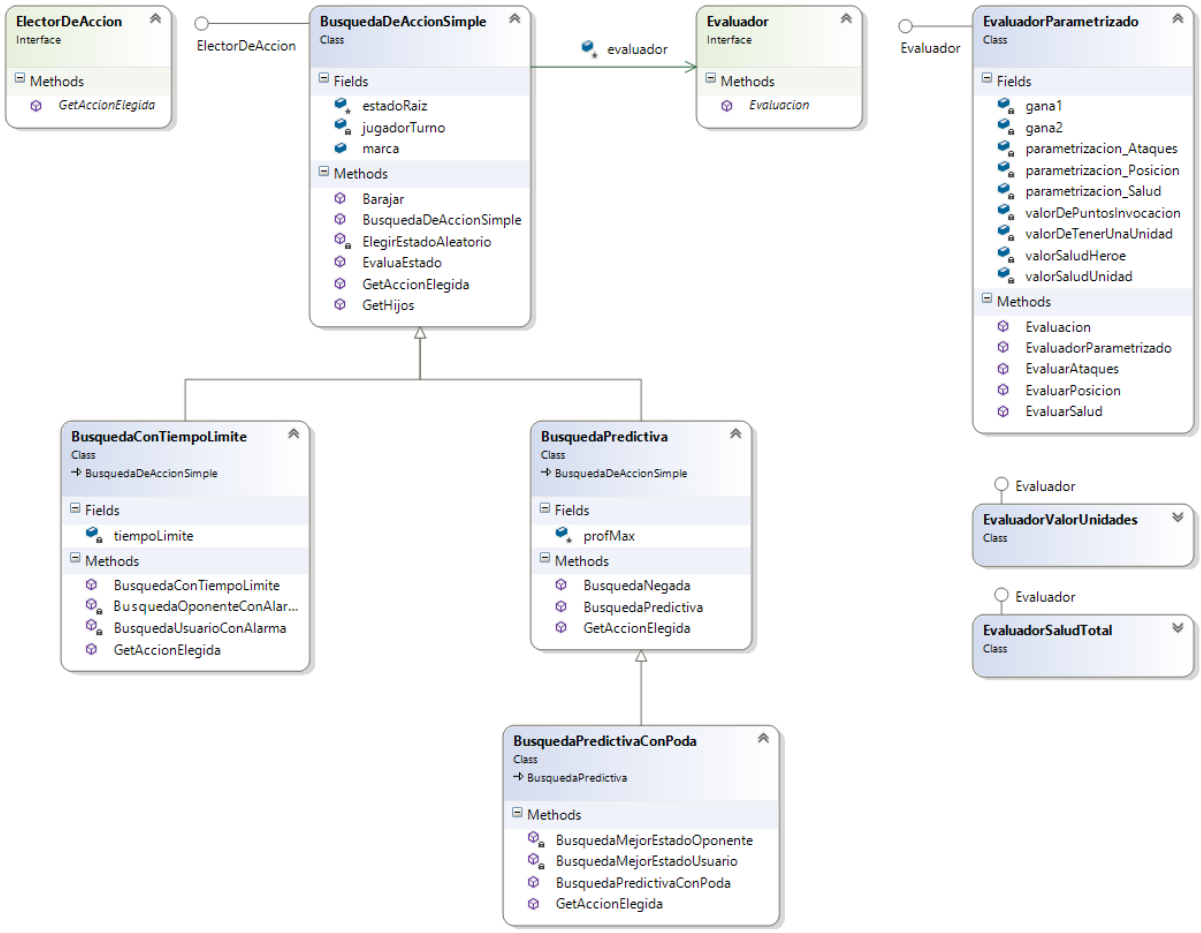


Figura 6.2: Diagrama final de la Inteligencia Artificial

6.6. Problemas y soluciones

Se han presentado problemas en cada una de las facetas de la implementación de la Inteligencia Artificial:

- En la evaluación se ha presentado el problema de la evaluación de las posiciones de las unidades.
    - Para ello se ha parametrizado la evaluación, lo que permitiría, en un futuro, encontrar los parámetros adecuados según la situación.
    - Otra posible solución sería el establecer estrategias de evaluación predefinidas, y disponer de un selector que elija la estrategia adecuada según ciertas características del estado.
- En el cálculo de hijos, el problema del factor de ramificación.
    - Una posible solución sería ejecutar el agente en el servidor, que tiene unos recursos acotados, sobre los que podemos trabajar. Obviamente supondría al jugador tener que estar conectado para poder jugar, y cargar más al servidor, tanto en cálculo como en comunicaciones.
- En la estrategia de búsqueda, no se ha encontrado un algoritmo claramente satisfactorio:
    - Para solucionarlo se podrían explorar otros acercamientos a la Inteligencia Artificial. Como pueden ser los árboles de comportamiento, el aprendizaje automático, u otros.

7. Conclusiones

El trabajo ha resultado exitoso gracias a varios puntos:

- Entorno de trabajo:
  Las herramientas y marcos de trabajo usados son de gran calidad, y disfrutan de una documentación y comunidad que ha supuesto una gran fuente de aprendizaje y ayuda.

- Trabajo en grupo:
  El trabajo en grupo se ha realizado de forma profesional y amigable, usando los principios de las metodologías ágiles, consiguiendo así una organización de las tareas y una comunicación, que en vez de generar problemas los ha mitigado.

- Buenas prácticas de programación:
  Antes de empezar a escribir la primera línea, como se ha visto en los diagramas de clase previos, se había pensado la arquitectura del sistema, usando patrones de diseño, el principal, que se ha usado varias veces es el **patrón de delegación**, que nos ha permitido dotar al sistema de toda la flexibilidad necesaria.

7.1. Aprendizaje

El trabajo ha supuesto la implementación de un sistema bastante extenso (más de 2.000 líneas de código contenidas en aproximadamente 70 clases), con una complejidad de herencia de 5 (es decir que una clase puede tener cuatro grados de herencia superior hasta llegar a la clase base primitiva). Para manejar todo esto, ha sido necesario mantener una arquitectura estructurada del sistema, tal y como se ha ido explicando a lo largo de la memoria y con las buenas prácticas de programación anteriormente mencionadas. Como alumno, esto ha sido muy beneficioso, ya que la razón de ciertos conceptos y enseñanzas aprendidas sale a la luz realmente en proyectos de cierta envergadura.

Un ejemplo de esto es la arquitectura Modelo Vista Controlador, que es una idea troncal en este proyecto, en el que en todo momento se ha hablado de la programación de la lógica de juego, que encapsula la estructura de datos y las funciones del juego, y los elementos visuales (vista). Ya que de no haber estructurado de esta forma el sistema, la vista habría quedado ligada a las funcionalidades internas del sistema, lo que habría causado graves problemas en ciertos momentos del desarrollo que ni siquiera han supuesto un leve impedimento. Otro ejemplo son las enseñanzas de la Ingeniería del Software sobre trabajo en equipo y sobre Análisis de

requisitos y Modelado, ya que, pese a que el sistema final difiere en la implementación interna (las clases tienen mucha más complejidad que las inicialmente planteadas), la estructura ha sido la que se planteó desde la fase inicial del desarrollo tras el análisis y el modelado. Esto ha supuesto el no llegar a callejones sin salida en cuanto a programación se refiere, y tener en todo momento claro donde ubicar cada funcionalidad del sistema.

- **Tecnologías**
  Este trabajo ha supuesto el aprendizaje en profundidad de ciertas herramientas, como pueden ser **Visual Studio** o **Unity** pero también el lenguaje de maquetación LaTeX para la elaboración de la memoria. Y el uso del lenguaje de programación **C#**, que pese a ser muy parecido a otros lenguajes orientados a objetos usados en el grado, ofrece ciertas particularidades interesantes, como puede ser el uso de **LINQ (Language Integrated Query)**[10] y el manejo tan simple y eficiente de colecciones que proporciona.

- **Fallos**
  El principal fallo ha sido subestimar el trabajo que necesita el desarrollo de un videojuego, más allá de la programación. Para desarrollar la parte visual y de interacción con el usuario que se espera de un videojuego, los esfuerzos dedicados han sido más tediosos, y han tenido un peor resultado, de lo que se pensaba en un inicio. Por otra parte, las reglas del juego tenían fallos de base, tanto para la experiencia de juego como a la hora de implementar un enemigo inteligente. Esto ha supuesto tener que eliminar o rehacer ciertos elementos del proyecto, no porque estuvieran implementados incorrectamente, sino debido a que no eran divertidos. Lo cual en un trabajo de este tipo es muy negativo, al tener que dedicar esfuerzos en vano por no disponer de unos conocimientos lo suficientemente sólidos de diseño de videojuegos.

7.2. Trabajo futuro

En todo momento se ha hablado de que el resultado de este proyecto era un producto mínimo viable, y por ello es un subproducto de la visión que se tenía de él, que esperamos poder completar. Debido a la estructura del sistema, implementar nuevas mecánicas de juego será algo bastante sencillo, y durante el trabajo se han visto bastantes posibilidades y cambios de las propias reglas que podrían dar lugar a un producto mucho mejor. También sería interesante estudiar la aplicación de las soluciones que se enumeran en la última sección del capítulo **Inteligencia Artificial**, o investigar soluciones alternativas.

Bibliografía

[1] Entertainment Software Association. Lo esencial sobre la industria del videojuego, 2016. http://essentialfacts.theesa.com/Essential-Facts-2016.pdf, visitado por última vez el 22/06/2017.

[2] Unity. Manual de unity. https://docs.unity3d.com/Manual/index.html, visitado por última vez el 22/06/2017.

[3] Kottwitz, Stefan. Editorial Packt Publishing. LaTeX Begginer's Guide. Marzo 2011. ISBN: 978-1-847199-86-7.

[4] Microsoft. Referencia de c#, 2017. https://docs.unity3d.com/Manual/index.html, visitado por última vez el 07/03/2017.

[5] Unity. Portal de la herramienta Unity collaborate, 2017. https://unity3d.com/es/services/collaborate, visitado por última vez el 22/06/2017.

[6] Documentación sobre manejo de casillas hexagonales. http://www.redblobgames.com/grids/hexagons/, visitado por última vez el 01/05/2017.

[7] Recurso del código externo pathfinder. https://www.assetstore.unity3d.com/en/#!/content/50282, descargado el 03/05/2017.

[8] Información sobre gameobject y monobehaviour. http://www.programering.com/a/MDO4UDNwATY.html, visitado por última vez el 23/06/2017.

[9] Ejemplo de manejo de colisiones en unity. http://pievisdev.blogspot.com.es/2015/05/survival-shooter-in-unity.html, visitado por última vez el 22/06/2017.

[10] Microsoft. Documentación sobre LINQ, 2017. https://msdn.microsoft.com/es-es/library/bb397926(v=vs.110).aspx, visitado por última vez el 22/06/2017.

[11] Ley de regulación de juego, 27 de Mayo de 2011.
http://www.boe.es/buscar/act.php?id=BOE-A-2011-9280, visitado por última vez el 22/06/2017.

[12] KENNEY group. Galeria de recursos para videojuegos abierta a uso público.
https://kenney.nl/assets, recursos descargados el 01/05/2017.

[13] Recurso de la imagen de la ficha guerrero.
https://pixabay.com/es/viking-guerrero-masculina-muscular-2009503/, recurso descargado el 19/05/2017.

[14] Recurso de la imagen de la ficha mago.
https://opengameart.org/content/necromancer, recurso descargado el 19/05/2017.

[15] Recurso de la imagen de la ficha arquero.
https://www.flickr.com/photos/jade_lilly/8267232152, recurso descargado el 19/05/2017.

[16] Recurso de la imagen de la ficha héroe.
http://jeradsmarantz.blogspot.com.es/2011/02/sucker-punch-designs.html, recurso descargado el 19/05/2017.

Anexos

Anexo I: Reglas del juego

Índice

I. Elementos del Juego

I.I. El tablero

El tablero está formado por 73 casillas hexagonales distribuidas como se muestra en la Figura 1

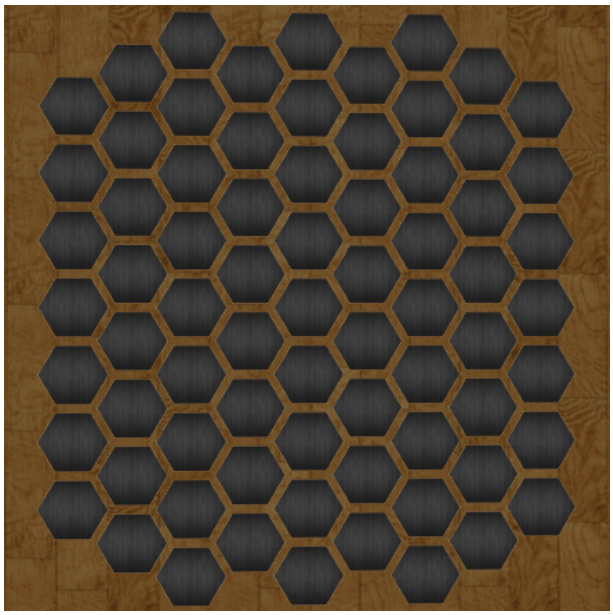


Figura 1: Tablero

Cada casilla puede, tener una única unidad asociada, o no tener unidad asociada. Cada unidad deberá tener en todo momento, una única casilla asociada obligatoriamente. A las casillas se les puede establecer si se puede pasar, o no, a través de ella, y cuantos puntos de movimiento cuesta hacerlo.

I.II. Los jugadores

En cada partida participarán 2 jugadores, numerados como Jugador 1 y Jugador 2. A los cuales se les asignará los siguientes colores, azul para el 1 y naranja para el 2, y se usarán durante el juego para distinguir a cada oponente.

Héroe del jugador

Ambos tendrán una unidad especial (Héroe) sobre el tablero de forma inicial, en la columna central del tablero.El objetivo, por tanto, de cada jugador, es el de eliminar a la unidad especial, que llamaremos Héroe, del jugador enemigo.

Unidades del jugador

Además del héroe, los jugadores podrán tener hasta 6 unidades extra en el tablero.

I.III. Las unidades

Valores de las unidades

Cada unidad contará con una serie de valores que determinarán su jugabilidad durante la partida, tales como:

-Vida: Al llegar a 0 la unidad muere.

-Ataque: Puntos de vida que restará a una unidad enemiga al atacar.

-Defensa: Daño que evita la unidad al recibir un ataque

-Rango de ataque: Distancia a la cual una unidad puede realizar un ataque.

-Rango de Movimiento: Distancia por la que se podrá mover como máximo durante un turno.

-Habilidades: Habilidades (activas o pasivas) que tiene una unidad.

-Puntos de acción: Número de veces que puede hacer una acción la unidad por turno, como atacar, o usar una habilidad.

-Puntos de invocación: Puntos de invocación necesarios para ponerla en juego.

Movimiento

Cada unidad cuenta con unos puntos de movimiento y de acción máximos por turno. Cada movimiento a una casilla adyacente, reduce el número de puntos de movimientos el coste de movimiento de dicha casilla, en la mayoría de casos, las casillas tendrán un coste 1, con lo que los puntos de movimiento muestran cuántas casillas se puede mover la unidad.
Para moverse a una casilla, por lo tanto, es necesario que la cantidad de puntos de movimiento sea igual o mayor al coste de dicha casilla.

Ataque y habilidades

Al atacar o usar una habilidad, automáticamente los puntos de movimiento pasan a 0, ya que no podrá moverse más en ese turno.
Los puntos de acción son la cantidad de ataques o habilidades que la unidad puede usar en el turno. En la mayoría de casos será 1, con lo que la unidad no podrá hacer ninguna acción de ningún tipo tras atacar o usar una habilidad.

II. El juego

II.I. Los turnos

Acciones del jugador en cada turno

En cada turno, el jugador podrá, realizar varias acciones, que se pueden clasificar de la siguiente forma:

- Posicionar una nueva unidad en el tablero:
  Cada jugador tendrá, como se muestra en la Figura 2, su zona de posicionamiento (ZP), que corresponde con las seis casillas adyacentes al Héroe.

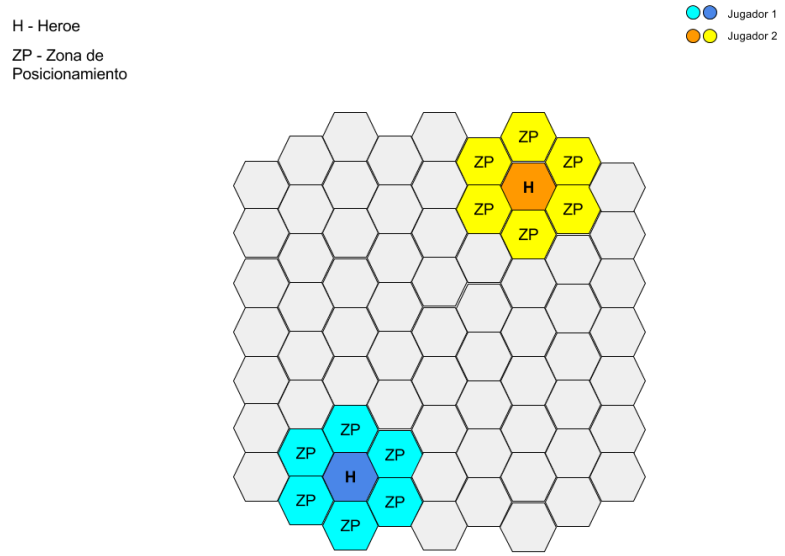


Figura 2: Zonas Invocación

Para posicionar una nueva unidad será necesario elegir una casilla vacía de la zona para poner ahí la unidad, que no se podrá usar hasta el turno siguiente.
Las condiciones de posicionamiento de unidad, así como el poder sustituir las unidades por evoluciones de estas, están indicadas en la Sección II.IV

- Mover una unidad disponible en el tablero:
  Cada unidad se podrá mover por el tablero, dependiendo de movimiento que tenga, para ello elegiremos la unidad, y se nos mostrarán las casillas disponibles a las que la podemos mover. Al hacer clic en una de esas casillas, moveremos allí la unidad.
  En cada turno, cada jugador podrá mover únicamente una unidad. Un ejemplo de la previsualización de movimiento se puede ver en la Figura 3

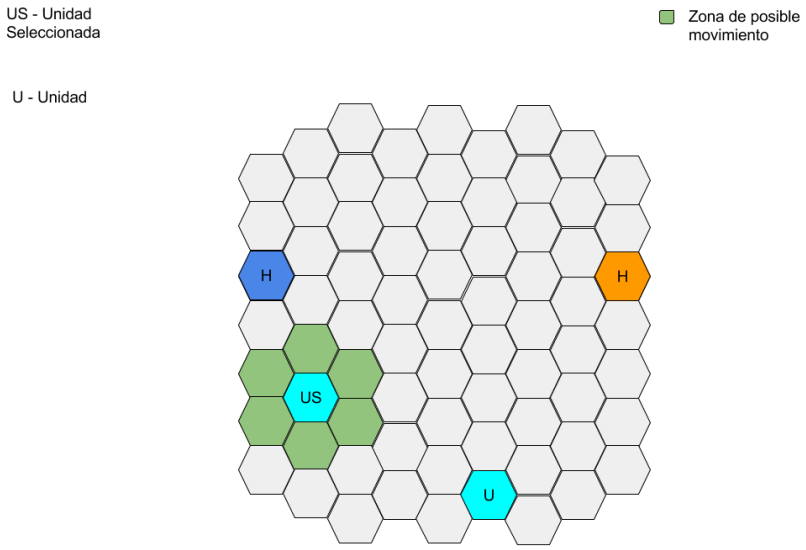


Figura 3: Zonas de Movimiento

- Ejecutar una acción de ataque de una unidad:
  Cada unidad podrá a atacar a otra enemiga, dependiendo de si está dentro del rango de ataque de la unidad, que puede variar, de la misma forma que el movimiento, se nos mostrarán las unidades enemigas atacables y podremos hacer clic en ellas para ejecutar el ataque. Un ejemplo de la previsualización de ataque se puede ver en la Figura 4

Duración de un turno

El turno termina cuando elegimos terminar el turno, cuando se acabe el tiempo, o cuando terminamos nuestras acciones.
El tiempo máximo de duración de un turno es de 30 segundos.

II.II. Eliminación de unidades y ataque

Cuando una unidad ataca a otra, la unidad atacada recibe puntos de daño de la siguiente forma:
Al recibir un daño X, si la unidad tiene 1 de defensa o más, el daño restará X puntos de defensa, hasta llegar a 0.

Al recibir ese daño X, si la unidad tiene una defensa igual a 0, se reducirán X puntos de vida a la unidad, hasta llegar a 0.

Si los puntos de vida de una unidad llegan a 0, dicha unidad es eliminada.

Ejemplos:

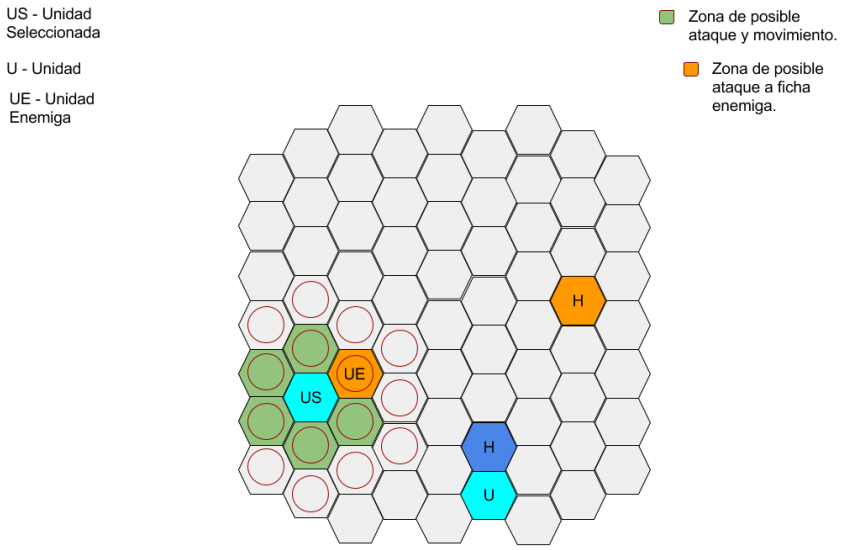


Figura 4: Zonas de Ataque

- Una unidad con vida 4, defensa 4, recibe un daño 3 – termina con vida 4 defensa 1.
- Una unidad con vida 4, defensa 1, recibe un daño 3 – termina con vida 4 defensa 0.
- Una unidad con vida 4, defensa 0, recibe un daño 3 – termina con vida 1 defensa 0.
- Una unidad con vida 1, defensa 0, recibe un daño 3 – es eliminada.

II.III. Jugador ganador de la partida

Cuando un jugador consigue eliminar al Héroe enemigo, dicho jugador es el ganador de la partida.

II.IV. Las Invocaciones

Invocación de unidad

Las invocaciones son las habilidades principales de los héroes, como se especifica anteriormente, cada jugador solo puede tener 6 unidades invocadas a la vez.

Invocación de Unidad Inicial

Las invocaciones son las habilidades principales de los héroes. Una vez por turno se puede invocar una unidad a elegir de entre las 3 disponibles de nivel 1.
Además, durante la partida únicamente se pueden realizar una cantidad de 10 invocaciones

iniciales
Las unidades únicamente podrán ser invocadas en las casillas circundantes al héroe.

Invocación por evolución

Se pueden usar puntos de evolución para sacrificar una unidad del tablero, eliminándola y sustituyendola por una evolución, (una unidad 1 nivel mayor a la sacrificada, que se encuentra en su árbol de evoluciones)
Cuando una unidad es invocada (inicial o evolución), no puede realizar ninguna acción durante el turno de su invocación (a menos que tenga una habilidad que se lo permita).

Obtención de puntos de evolución

Los Puntos de Evolución pueden ser obtenidos eliminando eliminando a las unidades enemigas del rival. Esta cantidad de puntos puede variar dependiendo de la unidad eliminada.

Árboles de Invocación

Cada unidad inicial contará con un árbol de evoluciones, el cual determinará el proximo tipo de invocación que se podrá realizar al evolucionar dicha unidad usando los puntos de evolución necesarios.
Cada nivel de profundidad del árbol indicará el nivel de las unidades, habiendo un máximo de profundidad 3, y por lo tanto nivel máximo 3.

El diagrama de la Figura 5 podría representar el árbol de evoluciones disponible para la unidad 1. Además, podría darse el caso de que el árbol de invocación de varias unidades distintas de nivel 1 converjan llegado a cierto nivel.

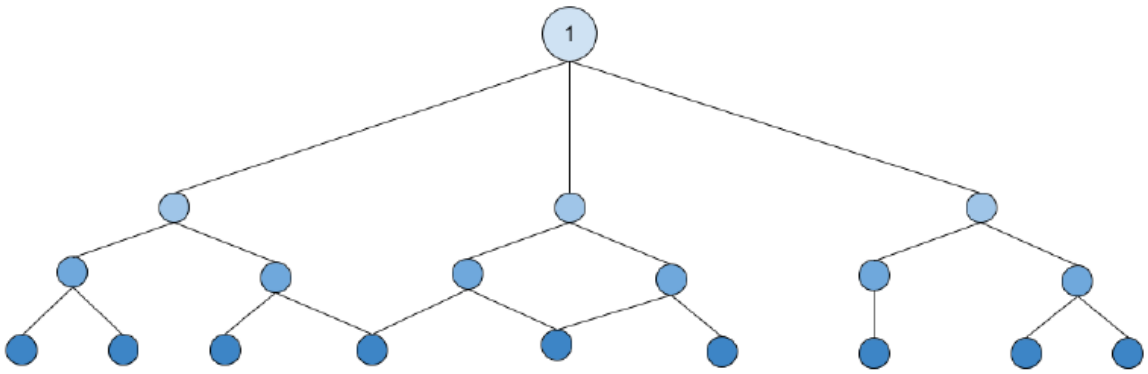


Figura 5: Ejemplo de árbol de evoluciones

III. Enumeración de unidades

III.I. Héroes

El héroe es la unidad más importante del jugador, ya que su muerte implica perder la partida. No obstante el héroe no es más que una unidad, que está situada en el tablero de forma inicial, con las siguientes particularidades:

- Durante la partida no se puede invocar ningún otro Héroe, es único.
- Tiene la habilidad "Invocar" que permite invocar una unidad de nivel 1.
- Inicialmente está en la columna central del tablero, de la fila superior o inferior, dependiendo del jugador al que pertenezca.

(En un futuro queremos que haya varios héroes, entre los cuales el jugador pueda elegir cuál quiere usar, y dependiendo del héroe podrá invocar distintas unidades de nivel 1, no obstante para el Trabajo de Fin de Grado hemos decidido que el héroe sea fijo, y el mismo para ambos jugadores, y por lo tanto tenga opciones de invocación fijas)
En el Cuadro 1 se pretende mostrar las valores base que tendrá la unidad héroe.
Significado de las cabeceras:

-Base: Unidad desde la que es evolucionada.

-Nombre: Nombre de la unidad.

-A: Ataque de la unidad.

-V: Vida de la unidad.

-D: Defensa de la unidad.

-R.M: Rango de movimiento de la unidad.

-R.A: Rango de Ataque de la unidad.

-P.A: Puntos de acción de la unidad.

-P.I: Puntos de invocación necesarios para invocar a dicha unidad, en el caso del héroe indican la cantidad de unidades que podrá invocar el jugador durante toda la partida.

-Rec: Puntos de invocación que se obtienen como recompensa por elimiar a dicha unidad.

-T.A: Tipo de ataque de la unidad.

-Vel: Velocidad de la unidad.

Nombre	A.	V.	D.	R.M.	R.A.	P.A.	P.I.	Rec.	T.A.	Vel,
Héroe	1	30	5	2	1	2	10	0	0	3

Cuadro 1: Héroe.

III.II. Unidades Iniciales

En el Cuadro 2 se pretende mostrar las unidades que pueden ser invocadas por el jugador en las casillas adyacentes al héroe.

Nombre	A.	V.	D.	R.M.	R.A.	P.A.	P.I.	Rec.	T.A.	Vel.
Guerrero	1	2	2	2	1	2	1	1	0	3
Arquero Aprendiz	1	1	1	3	2	3	1	1	0	3
Mago Aprendiz	2	1	0	1	3	3	1	1	0	3

Cuadro 2: Unidades Iniciales.

III.III. Evoluciones

En el Cuadro 3 se pueden observar las unidades de nivel 2 a las que es posible evolucionar las unidades iniciales.
Así mismo, en los cuadros 4, 5 y 6 se pueden observar las evoluciones de nivel 3 según la rama de guerrero, arquero y mago respectivamente.

Base	Nombre	A	V	D	R.M.	R.A	P.A.	P.I	Rec.	T.A.	Vel.
Guerrero	Guerrero n2 tanque	2	1	0	1	3	3	1	1	0	3
Guerrero	Guerrero n2 ataque	3	2	2	2	1	2	2	2	0	3
Guerrero	Guerrero n2 mov. ataque	3	2	1	3	1	3	1	1	0	5
Arquero Aprendiz	Arquero n2 armadura	1	3	1	1	3	1	2	2	1	4
Arquero Aprendiz	Arquero n2 ataque	3	2	0	1	3	1	2	2	1	4
Arquero Aprendiz	Arquero n2 mov.	1	2	0	2	3	2	1	1	1	4
Mago Aprendiz	Mago n2 armadura	2	2	2	1	2	1	2	2	2	2
Mago Aprendiz	Mago n2 ataque	4	2	0	1	2	1	2	2	2	2
Mago Aprendiz	Mago n2 armadura	2	2	2	1	3	1	2	2	2	2

Cuadro 3: Unidades Nivel 2.

Base	Nombre	A	V	D	R.M.	R.A	P.A.	P.I	Rec.	T.A.	Vel.
Guerrero n2 tanque	Guerrero n3 tanqueta	1	5	8	1	1	1	4	3	0	2
	Guerrero n3 tanque daño	3	3	5	1	1	1	4	3	0	2
Guerrero n2 ataque	Guerrero n3 lancero	3	2	2	2	2	2	4	3	0	3
	Guerrero n3 tanque daño	3	3	5	1	1	1	4	3	0	2
Guerrero n2 mov. ataque	Guerrero n3 mov. ataque ataque	5	2	1	3	1	3	4	3	0	5

Cuadro 4: Unidades Nivel 3 Rama Guerrero.

Base	Nombre	A	V	D	R.M.	R.A	P.A.	P.I	Rec.	T.A.	Vel.
Arquero n2 armadura	Arquero n3 armadura rango	1	3	1	1	4	1	4	3	1	4
	Arquero n3 armadura ataque	1	3	1	1	4	1	4	3	1	4
Arquero n2 ataque	Arquero n3 armadura ataque	1	3	1	1	4	1	4	3	1	4
	Arquero n3 ataquisimo	5	2	0	1	3	1	4	3	1	4
Arquero n2 mov.	Arquero n3 mov. ataque	2	2	0	2	3	2	4	3	1	4
	Arquero n3 mov. def.	2	2	2	2	3	2	4	3	1	4

Cuadro 5: Unidades Nivel 3 Rama Arquero.

Base	Nombre	A	V	D	R.M.	R.A	P.A.	P.I	Rec.	T.A.	Vel.
Mago n2 armadura	Mago n3 armadura ataque	4	2	2	1	2	1	4	3	2	2
	Mago n3 armadurisima	2	3	6	1	2	1	4	3	2	2
Mago n2 ataque	Mago n3 armadura ataque	4	2	2	1	2	1	4	3	2	2
	Mago n3 ataquisimo	7	2	0	1	2	1	4	3	2	2
Mago n2 rango	Mago n3 rango ataque	4	2	0	1	3	1	4	3	2	2
	Mago n3 rango rango	2	2	0	1	5	1	5	3	2	2

Cuadro 6: Unidades Nivel 3 Rama Mago.

Anexo II: Análisis de requisitos

Índice

IV. Actor objetivo del software: Usuario Jugador

Perfil: Persona de cualquier edad que usa la aplicación con la finalidad de jugar una partida.
Objetivo: Obtener un entretenimiento con el juego.

V. Requisitos Funcionales

V.I. El jugador puede:

1. Registrarse.
    1.1. Rellenará un formulario de datos obligatorios.
    1.2. El sistema guardará el nuevo usuario.
2. Identificarse.
    2.1. Introducirá los datos necesarios para identificarse.
    2.2. El sistema verificará los datos.
3. Ver un listado de las unidades del juego.
    3.1. Puede ver la información detallada de cada una.
4. Seleccionar el héroe que quiere usar.
5. Iniciar una partida.
    5.1. Iniciar una partida en linea.
        5.1.1. El sistema emparejará a los jugadores.
    5.2. Iniciar una partida de un jugador.
        5.2.1. El sistema creará una partida local contra la IA.
    5.3. El sistema creará una partida nueva.
        5.3.1. El sistema posicionará el héroe elegido por cada jugador en su lado del tablero.
        5.3.2. Establecerá el turno inicial de forma aleatoria.
6. Invocar unidades de apoyo
    6.1. El sistema proporcionará las unidades disponibles para invocar.
        6.1.1. El jugador puede elegir cuál de estas invoca.
    6.2. EL sistema establecerá las casillas disponibles para posicionar la unidad.
        6.2.1. El jugador puede elegir donde la posiciona.
7. Realizar acciones con una unidad durante la partida.

7.1. Puede seleccionar una unidad propia.

7.1.1. El sistema mostrará las acciones disponibles para dicha unidad.

7.2. Mover una unidad.

7.3. Atacar con la unidad seleccionada a una unidad enemiga.

7.3.1. Puede eliminar una unidad enemiga.

7.4. Evolucionar la unidad seleccionada.

7.4.1. El sistema mostrará las posibles evoluciones.

7.4.1.1. El jugador elegirá a que unidad evolucionará.

7.5. Puede cancelar la acción.

8. Cambiar de turno.

9. Rendirse.

10. Ver información de la partida.

10.1. Puede ver información detallada sobre las unidades en juego.

10.2. Puede ver información sobre los jugadores.

V.II. El sistema:

1. Detectará el final del turno.

1.1. Detectará si el jugador quiere cambiar de turno.

1.2. Detectará si al jugador no le quedan acciones disponibles.

1.3. Detectará si se ha acabado el tiempo del turno.

1.4. El sistema cambiará el turno al jugador contrario.

2. Detectará el final de la partida.

2.1. Detectará si un jugador derrota al héroe rival.

2.2. Detectará si un jugador decide rendirse.

2.3. El sistema mostrará en pantalla el resultado de la partida.

2.4. El jugador podrá volver al menú principal.

VI. Requisitos No Funcionales

VI.I. Requisitos de Aspecto

Interfaz

1. - Se establecerán formularios para registrarse e identificarse.
2. - El usuario podrá ver una representación del juego sobre el tablero (unidades aliadas y enemigas).
    2.1. - También se deberá poder visualizar e identificar las acciones de las unidades.
3. En la partida, las casillas y unidades serán objetos seleccionables.
4. Cuando se seleccione una unidad debe ser visible la siguiente información:
    4.1. Toda la información de dicha unidad.
    4.2. Todas las casillas a las que se puede mover y todas las unidades y/o casillas en rango de ataque.
    4.3. Una visualización con la información básica de todas las unidades a las que se puede evolucionar dicha unidad (si tiene evoluciones) o en caso del héroe, de las unidades que puede invocar.
5. Todas las unidades (héroe incluido) sobre el tablero de juego deben tener visibles su vida, defensa y ataque en todo momento.
6. Durante la partida deberá ser visible los puntos de evolución de los jugadores así como sus puntos de invocación restantes.

El estilo del producto

1. La aplicación diferenciará a los jugadores durante una partida con colores Azul (jugador) y Naranja/Amarillo (rival).
2. La fuente utilizada será Arial.
3. Se emulará un juego de tablero de madera.

VI.II. Requisitos de Facilidad de Uso y Aprendizaje

Facilidad de uso

1. La interfaz de usuario debe adaptarse a las acciones de la partida. Al pasar el ratón por encimad e los elementos de juego aparecerá información sobre ellos.
2. El Español será el idioma usado en el juego.

3. La aplicación debe verse correctamente en cualquier tamaño de pantalla y resolución.

4. EL sistema debe poseer interfaces gráficas dinámicas. Algunos botones cerrarán y abrirán paneles.

5. La distancia entre elementos de la interfaz deben tener una separación suficiente para que todo el conjunto se distinga con claridad.

Facilidad de aprendizaje

1. El sistema debe poder ser usado correctamente después de la primera partida.

2. El tiempo de aprendizaje básico del sistema por un usuario deberá ser menor a 2 horas.

VI.III. Requisitos de Funcionamiento

Requisitos de Velocidad

1. Toda acción es respondida por el servidor en un tiempo máximo de 5s.

2. Las acciones de juego deben ser respondidas en un tiempo máximo de 2s.

3. La inteligencia artificial realizará las acciones en un tiempo máximo de 10s.

4. La rapidez de la búsqueda de partida multijugador en lina, dependerá de la cantidad de usuarios que quieran jugar esta modalidad.

Requisitos de Seguridad Crítica

1. El sistema revisará las acciones enviadas al servidor.

Requisitos de Fiablidad y Disponibilidad

1. El sistema debe estar disponible con un nivel de servicio temporal para los usuarios 7 días x 24 horas x 365 días al año.

2. En caso de enviar una acción no disponible al servidor, no la ejecuta.

Requisitos de Escalabilidad

1. El sistema debe usar de forma óptima los recursos de conexiones a la base de datos.

    1.1. La base de datos debe permitir ampliar su capacidad de almacenamiento.

2. El sistema debe tener una clara partición entre datos, recursos y aplicaciones.

3. El sistema debe poder ampliar su capacidad para la gestión de un incremento en la cantidad de usuarios e información.

VI.IV. Requisitos Operacionales

Entorno Físico

1. La aplicación será usada en ordenadores y no es portable, por lo cual será un entorno amigable.

Entorno Tecnológico

1. El sistema será usado por el jugador a través de la aplicación Unity.
2. El hardware necesario para ejecutarlo es de minimos requerimientos.
3. La aplicación podrá ser instalada en cualquier Sistema Operativo de Windows actual.

Soporte

1. Introducir nuevas unidades implica añadirlas a la Base de datos.
2. Introducir nuevas mecánicas de juego implica modificar el código del juego y del servidor.

VI.V. Requisitos de Mantenimiento y Portabilidad

¿Cuál es la dificultad de mantenimiento de este producto?

1. Al tratarse de un juego, es muy recomendable introducir nuevas unidades y héroes, así como ir comprobando la experiecia de juego de nuestros usuarios.

¿Existen condiciones especiales aplicables al mantenimiento de este producto?

1. Las actualizaciones del juego requerirán descargarse la nueva aplicación completamente.

Requisitos de portabilidad

1. La aplicación solo será utilizable desde un orndenador con SO Windows.

VI.VI. Requisitos de Seguridad

¿El sistema es fiable?

1. Todos los usuarios deben iniciar sesión en el sistema para poder utilizarlo.
    1.1. Los usuarios deben ser identificados y autenticados con los datos de usuarios en la Base de Datos.
2. El sistema debe desarrollarse aplicando patrones y recomendaciones de programación que incrementen la seguridad de datos.

VI.VII. Requisitos Culturales y Políticos

¿Existe algún factor especial sobre el producto que lo pudiera hacer inaceptable por motivos políticos?

El juego puede ser considerado como bélico, a pesar de no tener imágenes de violencia y ser apto para todos los públicos. Sin embargo, pertenece más al género de estrategia, o juego de mesa.

VI.VIII. Requisitos legales

El sistema no almacenará ningún dato personal de los usuarios como nombre, edad, sexo...

Jurisdicción

1. El Artículo 18.4 de la constitución establece: "La ley limitará el uso de la informática para garantizar el honor y la intimidad personal y familiar de los ciudadanos y el pleno ejercicio de sus derechos.
2. LA ley 11/1998, «Ley General de Telecomunicaciones» que regula la privacidad y servicios en la red.
3. La Ley 13/2011, de 27 de mayo, de regulación de juego[11].

Estándares a cumplir

1. El estándar ISO 8402-94 que esablece la calidad de data a auditorías.
2. El estándar ISO 9126 de IEEE que establece la calidad y mantenibilidad de un producto software.